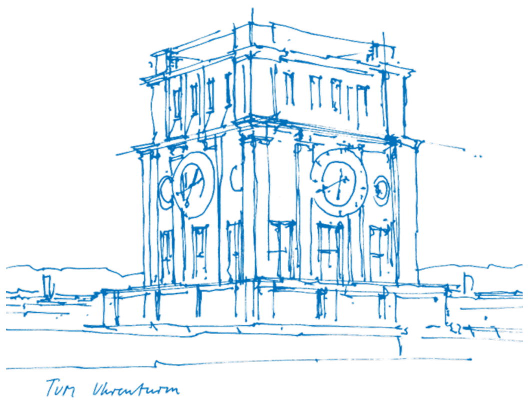
TUM School of Computation, Information and Technology
Technical University of Munich

Design, Implementation, and Evaluation of Data-driven Algorithms for Networking

Patrick Krämer, M.Sc.



Design, Implementation, and Evaluation of Data-driven Algorithms for Networking

Patrick Krämer, M.Sc.

TUM School of Computation, Information and Technology
Technische Universität München

Design, Implementation, and Evaluation of Data-driven Algorithms for Networking

Patrick Krämer, M.Sc.

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr.-Ing. Reinhard Heckel

Prüfer der Dissertation:
1. Prof. Dr.-Ing. Wolfgang Kellerer
2. Prof. Dr. rer. nat. Oliver Hohlfeld

Die Dissertation wurde am 26.04.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 26.09.2023 angenommen.

Abstract

(Communication) Service Providers (SPs) face increasing demand in data rate and high diversity in service requirements. To meet increasing demand and diverse services, SPs look towards a cloud-native approach to networking. Cloud-Native Communication Networks promise to meet increasing demand and satisfy diverse service requirements cost-efficiently by implementing Network Functions (NFs) as containerized microservices, enabling cloud-style automation and orchestration patterns for communication networks. The resulting Cloud-Native Communication Networks increase the network infrastructure's utilization and thus lower the cost of operation. Further, a cloud-native approach to networking shortens innovation cycles and speeds up service development, enabling distinguishing service quality and exploration of new business models.

To realize the benefits of Cloud-Native Communication Networks, specialized resource management, and packet processing algorithms are required. Specialized resource management and packet processing algorithms allow Cloud-Native Communication Networks to operate efficiently and achieve distinguishing service quality. Thus, the development of resource management and packet processing algorithms must keep pace with the shorter innovation cycles and faster service development. However, developing customized algorithms is challenging due to the large number of Containerized Network Functions (CNFs) resulting from the microservice-based implementation of NFs, and the need to incorporate use-case-specific properties into the algorithm design to make the algorithms efficient. Developing such algorithms takes time, effort, and expertise, potentially reducing the efficiency of Cloud-Native Communication Networks.

To match the development of algorithms to the complexity and agility of Cloud-Native Communication Networks, academia and industry look towards Machine Learning (ML) and Artificial Intelligence (AI). ML and AI can accelerate the algorithm design through the automatic extraction of patterns in data, allowing SPs to automate the specialization of resource management and packet processing algorithms to their specific use cases in an automated manner. Successfully designing, implementing, deploying, and maintaining ML-enabled communication networks is thus an important aspect for the operation of Cloud-Native Communication Networks.

This thesis presents approaches to automate the algorithm design for three prototypical networking use cases. As a first use-case, this thesis presents a microservice-based, ML-enabled traffic classification system deployable at the network edge that identifies the website a user accesses. As a second use case, this thesis shows an ML-enabled TE system that automates the design of a distributed TE mechanism for a given TE policy. As a third use case, this thesis presents an ML-enabled CNF platform that optimizes the co-location of CNFs on CPU cores, resulting in improved resource utilization. This thesis shows the feasibility of each application scenario through prototypical implementations and testbed evaluations. Finally, this thesis summarizes the gained experience from over 30 ML projects in networking into a process model providing guidelines to develop ML-enabled systems in the context of communication networks successfully.

Kurzfassung

Kommunikationsdienstleister (SPs) stehen einer steigende Nachfrage nach Bandbreite, sowie Diversifizierung von Anforderungen, die Dienstleistungen stellen, gegenüber. Um diese zu befriedigen setzen SPs zunehmend auf einen cloudbasierten Betrieb von Kommunikationsnetzen. Sogenannte cloudbasierte Kommunikationsnetze implementieren Netzfunktionen (NFs) mit Hilfe von Containertechnologie als Mikroservices, wodurch eine Automatisierung und Orchestrierung wie beim Betrieb bestehender Cloud Infrastruktur möglich wird. Diese Art des Betriebs verspricht durch eine erhöhte Ressourcenauslastung die steigende Nachfrage nach Bandbreite, sowie Anforderungen verschiedenster Dienste kosteneffizient nachzukommen. Zudem sollen cloudbasierte Kommunikationsnetze Innovationszyklen verkürzen, die Qualität von Dienstleistungen verbessern, und die Entwicklung neuer Dienstleistungen und Geschäftsmodelle beschleunigen.

Um eine hohe Effizienz herausragende Dienstleistungsqualität zu erreichen, benötigen cloudbasierte Kommunikationsnetze jedoch speziell angepasste Algorithmen für das Ressourcenmanagement und die Paketverarbeitung. Die Entwicklung solcher Algorithmen muss daher mit den kürzeren Innovationszyklen und dem erhöhten Tempo bei der Entwicklung und Verbesserung von Dienstleistungen Schritt halten. Die Entwicklung solcher Algorithmen wird jedoch durch die hohe Anzahl an containerbasierten Netzfunktionen (CNFs), die aus der Implementierung von Netzfunktionen als Mikroservices resultieren, und der Notwendigkeit, anwendungsfallspezifische Charakteristiken mit einzubeziehen, erschwert. Die Entwicklung solcher Algorithmen benötigt daher mitunter Zeit, Ressourcen und Fachwissen und reduziert daher möglicherweise die Effizienz mit der cloudbasierte Kommunikationsnetze betrieben werden können.

Um die Geschwindigkeit der Entwicklung von Ressourcenmanagement und Paketverarbeitungsalgorithmen auf die Geschwindigkeit und Vielfalt von cloudbasierten Kommunikationsnetzen anzupassen, rückt Maschinelles Lernen (ML) und Künstliche Intelligenz (KI) in den Fokus von Forschung und Industrie. ML und KI haben das Potential, die Entwicklung von Algorithmen durch die automatische Extraktion von Mustern aus Daten zu beschleunigen. Die erlaubt es SPs, Algorithmen automatisch an die jeweilige Gegebenheiten anzupassen. Das Entwickeln, Implementieren, Ausrollen und Betreiben von ML basierten Systemen für Kommunikationsnetze ist daher ein wichtiger Aspekt für den erfolgreichen Betrieb cloudbasierter Kommunikationsnetze.

Diesbezüglich präsentiert diese Dissertation Blaupausen für die Automatisierung der Entwicklung von Algorithmen für drei exemplarische Anwendungsfälle im Bereich der Kommunikationsnetze. Im ersten Anwendungsfall wird ein für den Betrieb am Rande eines Netzes auf ML und Mikroservice basierendes System für die Erkennung von Webseiten, die eine Nutzer besucht, vorgestellt. Als zweiter Anwendungsfall wird ein ML basiertes System zur Flusssteuerung vorgestellt, dass automatisch eine Flusssteuerung zu einer Gegebenen Flusssteuerung generiert. In einem dritten Anwendungsfall wird eine ML basierte CNF Plattform vorgestellt, die mit Hilfe von ML die Zusammenlegung von CNFs auf CPU Kerne optimiert und so die Systemauslastung erhöht. Abschließend wird, basierend auf den Erfahrungen von über 30 ML basierten Projekten im Bereich der Kommunikationsnetze, ein Prozessmodell für die Entwicklung von ML basierten Systemen für Kommunikationsnetze vorgestellt.

Acknowledgement

Completing this PhD journey has been a long and challenging road, and I couldn’t have reached this milestone without the support and encouragement of many wonderful people.

First and foremost, I want to express my deepest gratitude to my parents, and in particular, to my mother. Your unwavering love, support, and belief in me laid the strong foundation upon which I built this thesis. Your sacrifices, both big and small, have been the driving force behind my pursuit of knowledge, and I am eternally grateful for your guidance and inspiration.

I also want to extend my heartfelt thanks to my colleagues and friends who walked this path with me. Your camaraderie, shared knowledge, and countless discussions have been instrumental in shaping my ideas and motivating me to push the boundaries of my research. Your presence has made this academic journey not only intellectually stimulating but also incredibly enjoyable.

Throughout this journey, my wife has been my rock, providing unwavering support not only during the challenges of COVID-19 but also long before. You’ve been my sparring partner in countless discussions during lunch, and your insights have been invaluable in refining my ideas. You’ve stood by my side during the long nights and the stressful times of paper submissions, and for that, I am deeply thankful. Your belief in me, even when I doubted myself, has been the greatest gift of all.

To all the mentors, professors, and collaborators who guided me along the way, thank you for sharing your knowledge and expertise. Your insights have been invaluable in shaping my research and broadening my horizons.

In conclusion, this PhD journey has been a testament to the power of collaboration, love, and perseverance. I am humbled and honored to have had the support of such incredible individuals in my life. Thank you all for being a part of this remarkable journey.

Patrick Krämer, Stuttgart, September 24, 2023

Contents

Chapter 1

Introduction

(Communication) Service Providers (SPs) must improve the utilization of their infrastructure to meet the increasing demand in data rate and fulfill heterogeneous service requirements in a cost-efficient manner [1–4]. Improving the utilization requires the development of sophisticated resource management and packet processing algorithms. The increasing adoption of 5G and the rise of new services and applications such as eXtended Reality (XR), gaming, and video-based apps is expected to double the average data usage per smartphone from 22.4 GB in 2023 to 51.7 GB in 2028 [5]. In addition, applications and technologies like XR, Cyber-physical System (CPS), and Machine-to-Machine (M2M) communication require (ultra) low latency [4, 6]. Low latencies and high data rates are enabled through a Multi-access Edge Computing (MEC) infrastructure, i.e., geo-distributed compute locations allowing the deployment of services close to the user [2, 7, 8]. SPs look towards a cloud-native approach to networking to operate services on top of this infrastructure. Cloud-Native Communication Networks implement Network Function (NF) and services as containerized microservices, enabling cloud-style automation and orchestration patterns, e.g., through Kubernetes (K8S) [2, 3, 9]. SPs also hope to increase the development speed of new services and shorten innovation cycles to provide distinguishing quality and explore new business models [2, 9], thus helping them succeed in the increasingly competitive market [2].

At the same time, shorter innovation cycles and faster service development imply faster development of resource management algorithms. For example, new applications and services often require new Traffic Engineering (TE) mechanisms for optimal performance [10]. Similarly, a proliferation of Virtual Network (VN) and Service Function Chain (SFC) embedding algorithms exist for various use-cases [11, 12]. Aside from the need to tailor resource management to specific scenarios, Cloud-Native Communication Networks further complicate resource management through a large number of small Containerized Network Functions (CNFs), as monolithic Virtual Network Functions (VNFs) get split into containerized microservices [9]. However, resource management algorithms were already hard-pressed to scale to the level of complexity of networks before transitioning towards Cloud-Native Communication Networks [13]. Resource management often requires solving computationally hard Combinatorial Optimization Problems (COPs) such as bin-packing, load balancing, or flow problems [14, 15]. However, COPs are notoriously hard to solve exactly [16], and the time necessary to solve COPs often scales badly with the problem size [14, 15, 17]. Therefore, hand-engineered heuristics are usually developed and adapted for operations. Developing heuristics takes time and effort. The quality of the resulting resource management algorithms often depends on the ability to accurately model interactions between components in the network [18–20].

Moreover, new resource management algorithms might require developing and continuously adapting packet processing algorithms. For example, TE mechanisms make increasingly complex decisions based on network state, service, and communicating end-points [10, 21]. The service might not be directly available, e.g., in the presence of encryption [22–24]. To make the service available to resource management algorithms, packet processing algorithms classifying traffic online in the network might be necessary. Their development must keep pace with the deployment of new services. Besides, visibility about the service landscape might also be important for security reasons [22, 23] and business decision making [25].

Thus, matching the development speed of resource management and packet processing algorithms to the development speed of services to operate them efficiently will play an important role in an SP's success. To obtain packet processing and resource management algorithms that can operate live in the network, the industry and academia look towards data-driven methods, i.e., Machine Learning (ML) and Artificial Intelligence (AI) [2, 4, 13, 26]. ML and AI automatically identify patterns in data and encode them in algorithms [27, 28], promising a framework to automate and accelerate the development of resource management and packet processing algorithms. For example, ML can learn algorithms that map network state information into control decisions [15, 29]. Similarly, ML can automatically extract classification algorithms for TE mechanisms and application identification from observed traffic characteristics [24, 30, 31]. Specifically, ML allows the tuning of algorithms to a specific scenario [15, 17, 29, 30], e.g., tuning a resource management algorithm to the workload in a specific edge cloud or extracting the behavior of a new service. The resulting learned algorithms use a common set of instructions that allows acceleration in a standardized way, e.g., through instruction sets like the Advanced Vector Instruction (AVX) set, or hardware like Graphical Processing Units (GPUs), Tensor Processing Units (TPUs), and Field Programmable Gate Arrays (FPGAs).

However, ML projects come with their own set of challenges. For example, Gartner found that more than 50 % of ML models do not make it into production [32]. Venturebeat even reports 87 % [33]. Successfully designing, implementing, deploying, and maintaining ML-enabled communication networks is thus an important aspect for the successful operation of Cloud-Native Communication Networks.

This thesis presents data-driven resource management and packet processing algorithms for three use cases. The first use case is the development of an ML-enabled traffic classification system that identifies the website a user visits by observing the resulting encrypted traffic. The second use case is the development of an ML-enabled TE mechanism that learns the forwarding behavior of a TE policy in a Datacenter Network (DCN). The third use-case is the development of a ML-enabled platform for the efficient execution of CNFs with interference on shared resources. For each use case, this thesis presents the underlying challenge wrt. network operation derives a corresponding ML problem and develops the necessary ML models and data processing pipelines to automate the adaptation of ML models to different scenarios. To show the feasibility of the ML-enabled systems for each use case, the ML models are integrated into a networked system, and the resulting prototypes of ML-enabled systems are implemented and evaluated in a testbed. Finally, this thesis presents a process model that summarizes the gained experience and provides guidelines to successfully develop ML-enabled systems in the context of communication networks. Designing, evaluating, and implementing ML-enabled system results in many research challenges. The next section discusses some of these challenges.

1.1 Research Challenges

The data-driven design of resource management and packet processing algorithms has various research challenges:

- Determining the learned algorithm's input,
- preparing learning data,
- designing practical ML models,
- efficient implementation of ML models,
- Measurement-based evaluation of the learned algorithms.

This section discusses each of the research challenges. The next section will then present the thesis' main contributions.

Determining the learned algorithm's input. At a high level, an algorithm is a series of instructions that convert some input into some output. In the context of data-driven algorithm design, ML extracts these instructions from exemplary data. The user's task becomes providing input and output examples and ensuring that the output is computable from the input. Especially the specification of the input can become challenging. The input must contain all information that is relevant to explain the output. Depending on the type of algorithm that should be learned, specifying the input can require a deep understanding of the underlying optimization problem, network-specific concepts, and potentially internal workings of physical components.

If the learned algorithm should provide solutions to an underlying COP, understanding the underlying optimization problem is important because the problem defines how variables interact. This knowledge is required to decide on the features in the ML input. Thus, the underlying optimization problem must be defined formally or informally before the input and output can be specified.

Understanding the properties of network-specific concepts, e.g., protocols, application behavior, packet forwarding, monitoring interfaces, etc., is important because understanding them allows the formulation of a causal model, i.e., determining how the output might depend on potential inputs and which inputs are observable. This information is crucial to determine the input space.

If hardware-specific properties should be incorporated into the learned algorithm, then understanding the internal workings of physical components might be necessary. It is then necessary to know how specific system components impact the output variable and should thus be contained in the input.

In summary, just determining the inputs for ML can require expertise in optimization theory, networking domain knowledge, and a thorough understanding of the underlying hardware. Further, it might require the design, execution, and evaluation of experiments to establish the necessary understanding of the system.

Preparing learning data. Obtaining a good training set requires a thorough understanding of the later application scenario of the learned algorithm, i.e., potential modes of operations must be known. Collecting a sufficient data set can go beyond simply monitoring data during operation or generating random problem instances. To obtain a good training set, training samples for all potential modes of operation must be provided. This requires a careful design of the data generation process. If measurements are used for training, then the conduction of carefully designed experiments that result in the required conditions might be necessary. As a result, the nature of the data set used for training might look starkly different from what might be expected during operation.

Designing practical ML models. Modeling the data, i.e., learning algorithms from the examples, is difficult because it requires a deep understanding of the properties of different ML models and a careful design of models for a specific application scenario.

Deciding on ML models requires understanding the capabilities of different model classes. The choice of model depends thereby on the relationship between the inputs and outputs. Further, the choice of model is tightly coupled to the later application scenario, where the ML models might have to meet stringent performance requirements, e.g., wrt. inference speed and computational capacity. It is, therefore, not enough to choose any model and get the loss down but to always keep in mind how the ML model will later be embedded into the overall system.

Efficient implementation of ML models. After the ML models have been trained successfully, they have to be integrated into the systems they should extend. The implementation can be challenging if requirements for inference speed and computational demand are high. Implementing ML models efficiently requires a thorough understanding of the available computing hardware. For example, certain ML models can be accelerated through specific instruction sets, e.g., AVX, whose usage must be known to the developer. Reaping performance improvements from hardware accelerators might require, e.g., the batching of samples. Further, the ML model might allow an optimized implementation through its design or structure that the developer has to be aware of. Aside from model-specific optimizations, general efficient coding principles, such as using efficient data structures, optimizing memory access, etc., is necessary.

Assessing algorithm performance in measurements. The performance that the learned algorithms achieve in practice has to be determined through measurements. Simply measuring the execution time of the algorithms is thereby not enough. Instead, the algorithms must be integrated into the systems and evaluated there. This requires the setup of a corresponding test bed and the design and execution of measurements that evaluate the necessary performance indicators.

1.2 Contributions

This section summarizes the contributions of this thesis toward the data-driven design of algorithms for resource management and packet processing. Fig. 1.1 shows the structure of the thesis, the used methodologies and concepts, and the author's associated publications where these contributions were originally published. Source code and datasets are publicly available for transparency and as a contribution to the research community.

The first contribution is a process model for performing ML projects in the networking domain. The proposed process model extends established process models with the experience gained in over 30 ML-based projects in the networking domain. The thesis includes networking-specific aspects in the process model that are missing in existing ones. The resulting model might be relevant to other engineering fields beyond communication networks.

The second contribution of this thesis is an ML-enabled system for scalable traffic classification. Specifically, the system infers the websites a user visits from encrypted traffic. Traffic classification is an active area of research and

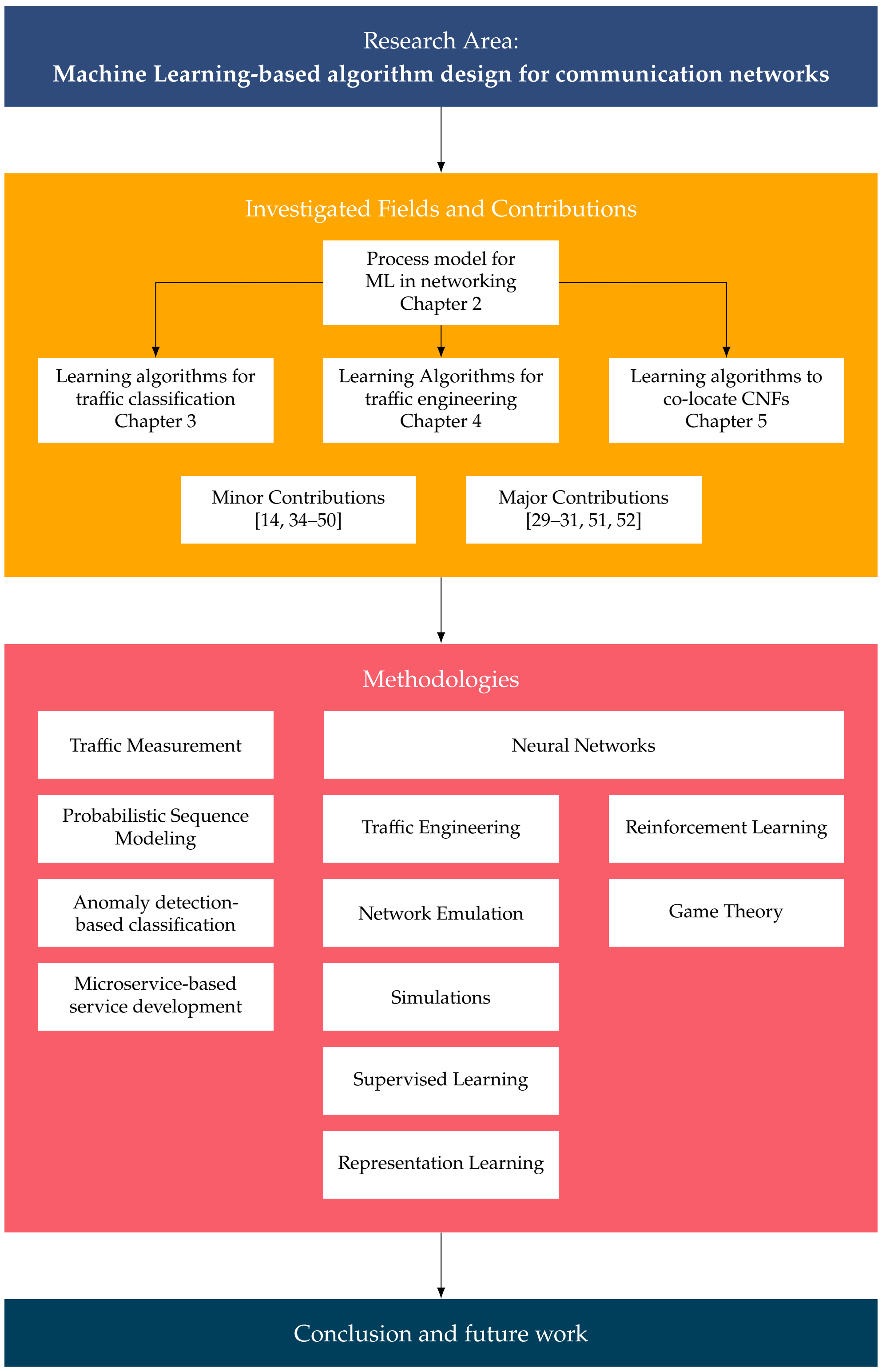

Research Area:
Machine Learning-based algorithm design for communication networks
Investigated Fields and Contributions
Process model for ML in networking Chapter 2
Learning algorithms for traffic classification Chapter 3
Learning Algorithms for traffic engineering Chapter 4
Learning algorithms to co-locate CNFs Chapter 5
Minor Contributions [14, 34–50]
Major Contributions [29–31, 51, 52]
Methodologies
Traffic Measurement
Neural Networks
Probabilistic Sequence Modeling
Traffic Engineering
Reinforcement Learning
Anomaly detection-based classification
Network Emulation
Game Theory
Microservice-based service development
Simulations
Supervised Learning
Representation Learning
Conclusion and future work

Figure 1.1 Graphical representation of the thesis' structure. The thesis first summarizes gained experiences into a process model. Then, the thesis designs, implements, evaluates, and deploys ML-enabled communication networks for three prototypical use-cases.

becomes increasingly challenging as the Internet adopts an encrypt-everything-regime. As a consequence, network operators might be oblivious to the network applications or, if possible, have to intrude into the network users' privacy, e.g., by acting as a proxy terminating the Transport Layer Security (TLS) connection. This thesis provides a compromise with an encryption-preserving classification system. The ML-enabled system has a cloud-native architecture, i.e., the system is implemented with microservices and could, e.g., be deployed at an edge cloud. The ML-enable system learns the required traffic classification algorithms from provided samples, using computationally efficient ML models for fast inference. Since frequent updates to the system are expected during the operational phase, the chosen ML models are fast to train, and the microservice-based architecture enables upgrades at runtime. The prototype implementation of the system can classify 424 website accesses per second at 10 Gbit/s and provides classification results with a median delay of a few 100 µs after the last required packet has been observed. This thesis presents the first prototype of a ML-enabled traffic classification system that can infer website accesses online and at line rate.

The third contribution of this thesis is an ML-enabled and end-host-based TE mechanism for Fat Tree DCN topologies corresponding, e.g., to an edge cloud. Tailoring TE to the specific applications deployed in a DCN can improve service quality and provide competitive advantages. The TE mechanism presented in this thesis automatically learns a TE policy's forwarding decision that can depend on the global network state from examples using a Neural Network (NN). To this end, this thesis proposes a Neural Architecture (NA) that can learn how to encode local state into update messages, which update messages are required to make decisions for a specific source-destination pair, and how update messages are mapped to forwarding decisions. Further, this thesis develops an efficient training algorithm exploiting special properties in forwarding decisions of TE policies. Together, data-driven TE mechanisms can be learned efficiently for a given TE policy in Fat Tree topologies. A prototype implementation shows that the resulting TE mechanism can react within 1.3 ms to changes in the network. Further, the prototype can be deployed in legacy networks and adapt the update message exchange to the traffic pattern in the DCN. This thesis presents the first working prototype of a NN-based TE mechanism.

The fourth contribution of this thesis is an ML-enabled CNF platform that improves resource utilization by learning load balancing and bin-packing heuristics for assigning CNFs to Central Processing Unit (CPU) cores. Improving a CNF platform's resource utilization is important to serve the increasing demand in data rates economically. To achieve this, the ML-enabled CNF platform uses ML to learn how CNFs interfere on cores. The ML-enabled CNF platform then uses the learned models to learn load-balancing and bin-packing heuristics with Reinforcement Learning (RL). To learn algorithms efficiently, this thesis proposes a novel training approach by linking Game Theory, RL, and combinatorial optimization in a novel way. The resulting learning approach allows the automatic tailoring of algorithms assigning CNFs to CPU cores to the CNFs and SFCs running in, e.g., an edge cloud. The prototype implementation of the system shows that the learned algorithms improve the packet processing efficiency by 58 %.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the process model for ML projects in networking. Chapter 3 presents the ML-enabled system for scalable data-driven traffic classification. Chapter 4 presents the ML-enabled TE mechanism. Chapter 5 presents the ML-enabled CNF platform. Finally, Chapter 6 summarizes this thesis and presents suggestions for future work.

Chapter 2

Applying Machine Learning in Networking

This chapter synthesizes the experience gained during more than 30 ML projects in the networking domain into the Machine Learning Applications in Networking (MaLANe) process model. MaLANe reflects in the chapters 3, 4, and 5. MaLANe provides a comprehensive blueprint for carrying out ML projects in networking, i.e., projects that consider the full life-cycle of designing, implementing, deploying, and operating ML-based applications in networking.

Successfully carrying out ML projects is challenging. A 2019 report of Venture Beat shows that only 13 % of Data Mining projects succeed [33], similarly, Gartner reports 2020 that 53 % of ML projects make it from prototype to production [53] - a number that increased only slightly to 54 % in 2022 [32]. The reason lies in the complexity that ML adds to software development. Traditional software projects are already complex, failure-prone, and require a broad range of expertise. ML adds additional collaboration points, requires new expertise, and different working styles [54]. The successful completion of ML projects in networking can be expected to be even more challenging. After all, ML projects in networking require additional expertise, mindsets, and collaboration points compared to purely software-based ML projects.

Experience shows that projects in the networking domain are highly interdisciplinary, requiring expertise in business, network engineering, data science, ML engineering, software engineering, and hardware engineering. In practice, the expertise is most likely provided through an interdisciplinary team, which results in a high risk for miscommunication and cultural clashes [54]. Integrating the knowledge from members with different backgrounds is a key challenge [55]. Further, the availability of high-quality data often is a problem in networking. Comprehensive data sets for ML projects are usually unavailable and must be created as part of the project, requiring a deep technical understanding of the networked system. Moreover, ML applications in networking have to cope with constrained computational resources frequently, putting an increased emphasis on the ML model choice and their implementation for operation [56–59].

MaLANe provides a blueprint for carrying out ML projects in networking, supporting novices and experienced analysts alike. MaLANe supports the integration of knowledge from persons with different technical backgrounds, supports the acquisition of high-quality data, gives advice on formulating quality and performance metrics towards ML models, and points out important aspects that should be documented and communicated explicitly. Further, MaLANe includes a perspective on scientific projects that usually have different requirements than projects with an initial business objective. MaLANe focuses on the networking domain and includes a perspective on RL. RL requires different activities than Supervised Learning (SL) and Unsupervised Learning (UL) on which previous process models focus.

MaLANe is especially helpful in academic projects. In academia, teams usually consist of students and young PhDs. Both are early in their careers and had little opportunity to gain a broad experience in data science projects. Further, in engineering fields, students and PhDs often have high-level data science skills since it is not the main focus of their studies. In such a setting, MaLANe can help avoid common pitfalls such as: Neglecting the acquisition of data, collecting the wrong data for the learning problem, neglecting to check the data quality, relying on data that is unavailable during operation, solving trivial or practically irrelevant problems, choosing models that cannot be integrated into networked systems, learning models that do not generalize in real systems or defining the wrong learning problem. MaLANe can guide these teams in planning and performing projects with impact.

This chapter is organized as follows. Sec. 2.1 introduces background information for process modeling, presents the basic process model for Data Mining, and presents related work. Sec. 2.2 presents the MaLANe process model and explains its activities. Sec. 2.3 closes this chapter and discusses the implication of this process on the application of ML in the networking domain.

2.1 Background and related work

This section introduces background information on process models and presents related work. Sec. 2.1.1 introduces background information and terminology concerning process modeling used throughout the thesis. Sec. 2.1.2 introduces the CRoss Industry Standard for Data Mining (CRISP-DM), forming the basis of MaLANe and being the most widely used process model to manage Data Mining and ML projects in 2014 [60]. Finally, Sec. 2.1.3 gives an overview of the State-of-the-Art (SoA) process models for Data Mining and ML projects.

2.1.1 Process Models

Modeling a process includes describing the activities of processes, required tools, documents, information, and necessary expertise. As a result, process models facilitate the division of labor that modern enterprises are based on [61]. As a result of the division of labor, people involved in a process perform specialized activities and might not be directly confronted with the full complexity of the process. A process model improves the understanding of the process among the involved people and organizations and helps to improve communication and identify and prevent problems [54, 61, 62].

A process is a chain of events, activities, and decision points. Each of which includes actors and objects. The activities lead to an outcome in which at least one customer has an interest. Events in a process have no duration and are things that happen automatically. Activities correspond to actions that must be performed and have a (finite) duration. Simple activities that consist of one well-defined unit of work are called tasks. Actors correspond to humans and non-human entities that perform tasks of an activity. Objects are passive elements that can serve as input and output to activities or are used during an activity. A customer is an entity that is interested in the process's output. Customers can be internal or external, i.e., correspond to entities within an organization or external entities [61].

In organizations, processes are key to competitiveness and economic success, especially in organizations primarily delivering services to their customers. An organization offering the same service as another has a competitive advantage if it implements superior processes, resulting in better service quality or cost savings [61]. This also holds for internal processes, for example, in ML projects. ML projects are complex, and their success depends on the right mix of tools and people with different backgrounds [62]. Process models for ML projects allow customers to have reasonable expectations towards the project, improve the project's execution, give structure to the documentation, support communication, and thus improve the overall quality [62, 63]. Given that many ML projects fail, a good process model that helps improve the success ratio of such projects can greatly improve an organization's competitiveness.

2.1.2 The CRISP-DM process model and its limitations

The CRISP-DM process model is the reference process model that MaLANe extends. Although CRISP-DM was proposed in the 1990s, CRISP-DM remains the most used process model [60]. The CRISP-DM model consists of six activities [62, 63]. Figure 2.1 shows the six steps in the CRISP-DM process.

1. *Business Understanding* defines well-defined goals and requirements and a clear task definition.

2. *Data Understanding* checks for available data and evaluates the data's quality and usability for the current project.

3. *Prepare Data* selects features from the data and potentially engineers new features to enrich the input for the Modeling phase.

4. *Train Model* uses statistical methods (including, but not limited to ML methods) to extract the previously defined knowledge from the gathered data.
5. *Evaluation* evaluates the tested models and selects the most suitable model.
6. *Deployment* prepares the results, presents them to stakeholders, and potentially deploys the model into production, i.e., automates the evaluation.

Each activity in the CRISP-DM process are sub-processes, i.e., consist of smaller units of work.

The CRISP-DM process model diverges from the textbook definition of a business process. In practice, an instance of the CRISP-DM process model is a highly iterative and creative process with many parallel activities. CRISP-DM instances can look markedly different from each other [54, 62]. Further, activities have well-defined start and end criteria in the usual business process definition. A subsequent activity only starts once the preceding activity finishes. However, experience shows that activities are not that clear cut in Data Mining processes. Here, the activities can overlap, and the order corresponds more to a precedence property, i.e., subsequent tasks can only start after the preceding ones have already started [62].

The CRISP-DM process was designed to be industry and application-independent. Unsurprisingly, CRISP-DM lacks essential aspects for successfully carrying out ML projects in networking. Experience shows four major aspects that are lacking:

1. CRISP-DM assumes data is gathered and available in databases.
2. The modeling step focuses on statistical modeling and is tuned toward SL.
3. CRISP-DM assumes the final model to be a pure software artifact.
4. CRISP-DM neglects the operational phase of the final application.

CRISP-DM includes data preparation as a step. However, it focuses on scenarios where data is available in databases. In networking, like engineering [64], the necessary data often does not exist in the first place. Before any data analysis and modeling tasks can start, necessary data must be gathered. This usually results in testbed measurements or measurements in the network during operation. Both are challenging tasks on their own.

CRISP-DM focuses more on detecting and modeling relationships between variables. Concerning ML, CRISP-DM is biased towards SL. However, in networking, RL is as important as SL, especially in the context of resource management. RL plays a crucial role in learning control algorithms for networked systems. RL has markedly different requirements for the preparation of data and the training of models than SL. CRISP-DM does not reflect these tasks.

The evaluation activity in the CRISP-DM process evaluates the performance of different models on test data [62, 63]. However, in networking, the models

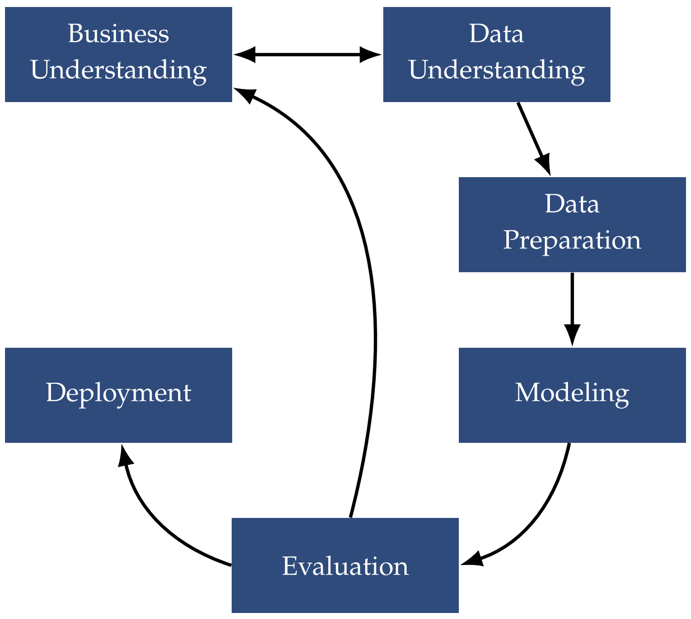


Figure 2.1 The CRISP-DM process steps.

must operate inside a networked system. Important is thus not only ML-related performance measures such as loss or accuracy but rather the overall performance of the ML-enabled networked system. Further, trained ML models usually cannot be deployed into the network directly. The models have to be implemented and integrated into the networked system first. CRISP-DM does not consider this step.

The output of a CRISP-DM process instance can range from a simple report to the implementation of a repeatable Data Mining process [62, 63]. However, in networking, the project's outcome is usually an ML-enabled application that operates inside the networked system. The operation of ML-enabled networked systems comes with additional maintenance tasks that are not reflected in the CRISP-DM process.

Thus, Sec. 2.2 proposes a re-design of CRISP-DM that addresses these and other shortcomings of CRISP-DM and its variants.

2.1.3 Related Work

Next to CRISP-DM, two other process models have widespread adoption: Knowledge Discovery in Databases (KDD) [65] and Sample, Explore, Modify, Model, Access (SEMMA) [66].

The KDD process model is the basis for CRISP-DM and SEMMA. KDD is a general process model for extracting knowledge from data with the help of analysis algorithms. KDD defines five activities [65]:

1. Selection of data for analysis.
2. Data-preprocessing and cleaning to obtain a consistent data set.
3. Data transformations for dimensionality reduction.

4. Data Mining for pattern discovery in the data.
5. Evaluation and interpretation of the identified patterns.

The activities in the KDD process are refined in various derived process models [67, 68].

SEMMA has been developed by the SAS Institute to guide users through a Data Mining software of SAS. As a consequence, SEMMA focuses narrowly on technical steps, corresponding roughly to the *Data Understanding*, *Prepare Data*, *Train Model*, and *Evaluation* activities of the CRISP-DM process model. That is, SEMMA does not include business understanding and the final deployment. Still, SEMMA is considered a general process model and used by 8.5 % of the participants in a KDnuggets poll [69].

KDD, SEMMA, and CRISP-DM form the basis for domain adaptations and extensions. Plotnikova et al. [67], and Martínez-Plumed et al. [68] provide comprehensive overviews of the various derivatives. From those, the Data Mining Methodology for Engineering Applications (DMME) [64] process model is closely related to MaLANe. In addition, the CRoss Industry Standard Process model for the development of Machine Learning with Quality assurance methodology (CRISP-ML(Q)) [70] and the Data Science Trajectories (DST) [68] process models share aspects with MaLANe as well.

DMME adapts CRISP-DM to the specific context of engineering applications. The most notable extensions are data generation and collection steps since data is not readily available in databases for many engineering applications. Further, DMME introduces an implementation stage concerned with rolling out the finished Data Mining method in a production environment. This explicitly includes training the workforce on the corresponding machines. MaLANe differs from DMME in that MaLANe includes a specific ML perspective that is lacking in DMME. Further, the tasks for data generation and especially deployment in networking differ from those covered in DMME. The DMME process model focuses on physical variables, for example, pressure, stress, deformation, etc. However, many problems arise in networking from COPs. Further, measurements in the networking domain often concern system variables and system behaviors. Those are often unknown before ML projects are started and require an agile approach. During deployment, DMME assumes that the resulting model runs separately from the machine under consideration. In networking, the resulting model must often be incorporated into the system, resulting in additional challenges.

CRISP-ML(Q) adapts CRISP-DM to ML specific projects, that have different requirements than pure Data Mining projects. Specifically, CRISP-ML(Q) identifies the deployment and continuous monitoring of the finished application as crucial but missing aspects. Further, CRISP-ML(Q) includes methods for quality assurance borrowed from other industrial process models such as Six Sigma [71] and Failure Modes and Effects Analysis (FMEA). MaLANe differs from CRISP-ML(Q) in that MaLANe explicitly covers the generation of data through measurements and the implementation of the ML model in networked systems. CRISP-ML(Q) assumes the data is mostly given, and the trained model is readily implementable as a software artifact, i.e., the model must be incorporated into an existing process rather than interfacing with other hardware components and systems.

DST distinguishes between Data Mining and data science. Martínez-Plumed et al. argue that previous process models apply to Data Mining and do not fully embrace the diversity of data science projects. The authors argue that previous process models are goal-oriented and sequential. In their DST model, the authors intend to capture the versatility of data science projects by defining exploratory activities, goal-oriented activities (taken from CRISP-DM), and data management activities. DST does not give transitions between the activities. Instead, the authors propose to choose the sequence according to the project at hand. The authors give examples of possible trajectories through this space that correspond to different types of data science projects. MaLANe differs from DST in that MaLANe provides clear guidance through the activities, giving stronger support, especially to novices. MaLANe balances rigidity and flexibility by emphasizing agile and iterative execution. Further, MaLANe includes a strong perspective on ML, which DST does not include.

Another line of research that is loosely related is the investigation of problems in Data Mining, data science, and ML projects [54, 55, 72]. The authors methodically evaluate interviews with practitioners and evaluate problems that hinder projects. The authors do not propose process models. However, their work gives important insights into problems that can arise in such projects. Process models should therefore incorporate these insights to overcome these problems and enable the smooth execution of data science projects.

2.2 The MALANE process model

Fig. 2.2 shows the major steps of the MaLANe process model. The process consists of ten steps:

1. Translate Objective,
2. Formalize Problem,
3. Analyze Network and System,
4. Generate Data,
5. Investigate Data,
6. Prepare Training,
7. Train Model,

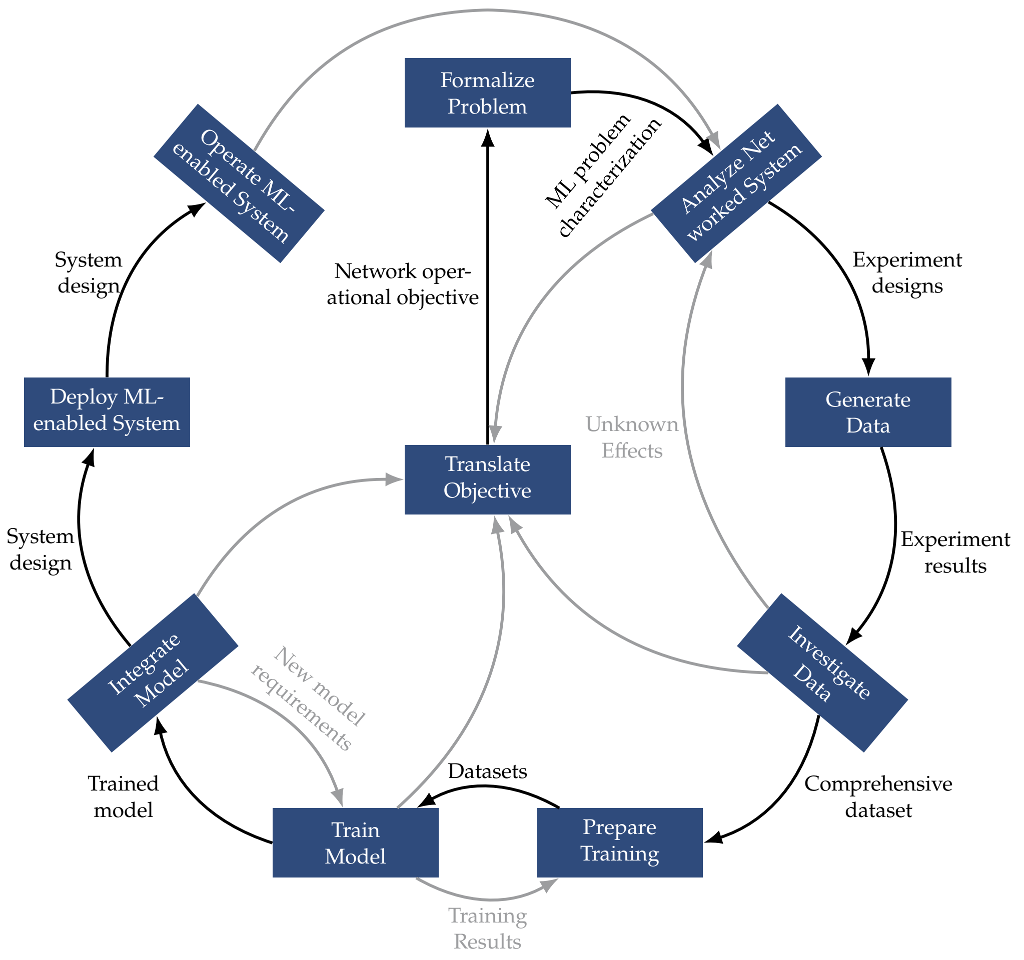


Figure 2.2 The Machine Learning Applications in Networking (MaLANe) process model. Boxes show MaLANe's activities and sub-process. Arcs correspond to transitions. Labels on the arcs to objects that pass between activities.

8. Integrate Model,
9. Deploy ML-enabled System,
10. Operate ML-enabled System.

The remainder of this section describes the individual steps and their respective expected outcomes.

2.2.1 The Translate Objective Activity

The *Translate Objective* activity is closely related to the *Business Understanding* activity in the CRISP-DM process model [62, 63], the *Define* step in Six Sigma [71], and the initial development activity in the IEEE Standard 1074-1995 for developing software life cycle processes [73]. The goal of the *Translate Objective* activity is uncovering the underlying network-specific objectives to a high-level business objective. In particular, the primary goal of the *Translate Objective* activity is to avoid producing the right answer to the wrong question. Experience and related work show that two variants exist for this activity: A clear goal-oriented variant and an exploration-oriented variant often observed in academia.

Goal-oriented. The *Translate Objective* activity has a business objective as input, and the possibility to meet this objective with the help of ML might have already been established. The goal of the *Translate Objective* activity is to translate the business objective into a network operation objective. The business and the operational objective should be measurable [71, 73]. Only measurable objectives allow a clear evaluation of the project's success. Further, the assumptions behind translating the business objective to the network operation objective should be documented. Making the assumptions transparent allows their verification in an early stage of the project.

A fictional example of a business objective could be the reduction of Operational Expenditures (OPEX) by 20 %. On the network operation level, the OPEX reduction could correspond to a reduction in power consumption by improving the resource utilization by 30 %. Information from monitoring systems indicates underutilized resources, i.e., the potential to improve resource utilization by at least 30 % exists. A rich body of literature shows that ML can solve resource allocation problems.

Exploration-oriented. Here, the business objective is exploratory, i.e., the objective might not express specific, measurable quantities. Instead, the business objective resembles an open question that should be answered during the project. For example, the business objective could be the identification of new ML-based business cases, the investigation if ML is applicable in a certain use-case, or the development of new data-driven business cases using the own operational data [54,

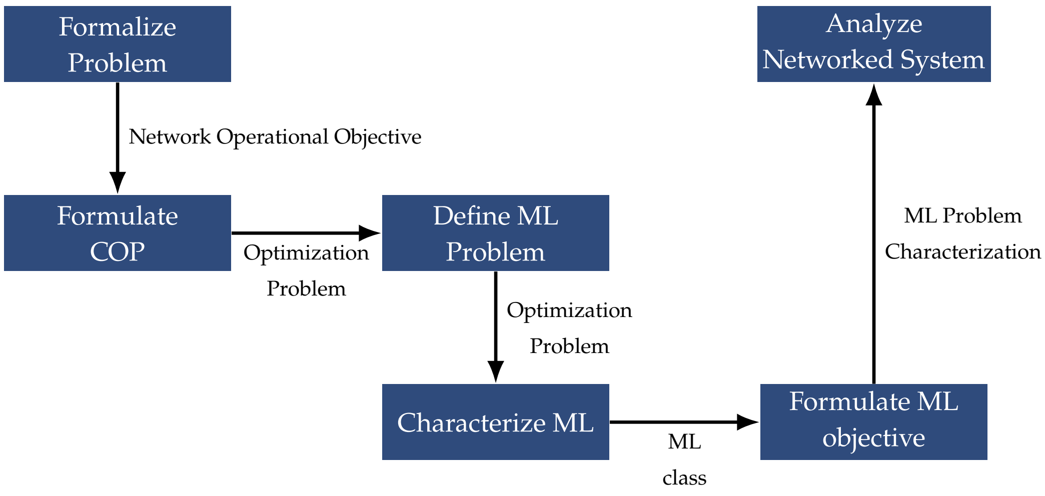


Figure 2.3 The *Formalize Problem* sub-process consists of two activities: *Formalize COP* and *Define ML Problem*. The latter consists of two sub-processes, *Characterize ML* and *Formulate ML objective*.

68]. While not as straightforward as in the goal-oriented variant, the exploration-oriented variant still needs a translation into network operation objectives, i.e., how the new business case or application scenario manifests itself in the network.

Projects in academia regularly fall into this category. The business objective often is feasibility studies, e.g., if it is feasible to design such a system and gain an advantage over existing systems while achieving better or competitive performance. The network operation objective is not given quantitatively but rather qualitatively. For instance: Achieve better performance than SoA, or: achieve at least competitive performance while providing a benefit in the ease of use.

Chapter 3 shows such an example. Here, the business case is to investigate if ML can realize distributed TE mechanisms that can operate in a network and achieve competitive performance.

2.2.2 Formalize Problem

The *Project Formalization* is unique to networking and represents a sub-process with two activities: the *Formalize COP* and the *Define ML Problem* sub-process. The *Formalize Problem* activity translates the network operational objectives from the *Translate Objective* activity into an ML problem characterization. ML engineers and network engineers collaborate in this activity to ensure that the resulting ML application will meet the business objectives.

Formalize COP. The *Formalize COP* activity relates to the COP that underlies many problems in networking and network operation in particular [14, 15]. The *COP Formalization* activity specifies, characterizes, and ideally mathematically formulates the COP.

Experience shows that COPs occur in many ML projects in networking in one of two forms: 1) The operational objective directly translates into a COP, and 2) the design decisions for the ML application, and later implementation depend on the solution of a COP. In the first case, ML learns algorithms that solve the COP. Here, the COP provides important information on variables in the systems, how variables interact with each other, and how variables interact with the optimization objective. This information is relevant for the design of the output space, the input space and thus statistics that must be observed, the qualification of the nature of interactions (linear vs. non-linear), and other ML model-related aspects. Further, the COP formulation can make missing information visible and assumptions explicit. For example, the nature and effect of interacting variables might be unknown. While a simplifying assumption on the interaction based on domain knowledge can be made, being aware of and documenting this assumption can become important if the resulting ML model does not meet quality requirements. Then, the assumption can be revisited, and an additional MaLANe process focusing on this interaction can be triggered to understand the interaction better. Potentially, even learning a data-driven model of the interaction, resulting in a better performing ML model. Chapter 5 will show such an example.

In the second case, ML does not learn an algorithm to the COP. Instead, during deployment, the ML-based algorithm might be deployed in the network based on the solution to a COP. An example is facility location problems that can arise for monitoring and traffic classification solutions. Here the sites at which monitoring systems are deployed must be decided, such that all or a certain fraction of the traffic is monitored [74]. The solution to this problem is important to judge the later requirements of the ML-enabled system and the ML model. For example, relevant information on the expected processing requirements can be deduced from the placement and expressed as clear performance objectives towards the used ML models. This kind of information is especially important for exploration-oriented projects. Chapter 3 will show such an example.

Explicitly including the COP formulation as a dedicated step in the MaLANe process model is based on experience. In most projects, one of the two cases was present. Formally modeling the problem thereby simplified the ML design and allowed the early identification of potential bottlenecks. Projects that did not include explicit modeling took longer to complete because relations were overlooked. The result was a poorly performing ML model or an ML-enabled system that could not operate in the network. Investigating the cause of the poor performance often proved difficult and time-consuming. Further, knowing the problem's nature enables the focused search in the related work for potential heuristics and previous approaches that can serve as a baseline to assess the ML model's quality and to model and solve the problem.

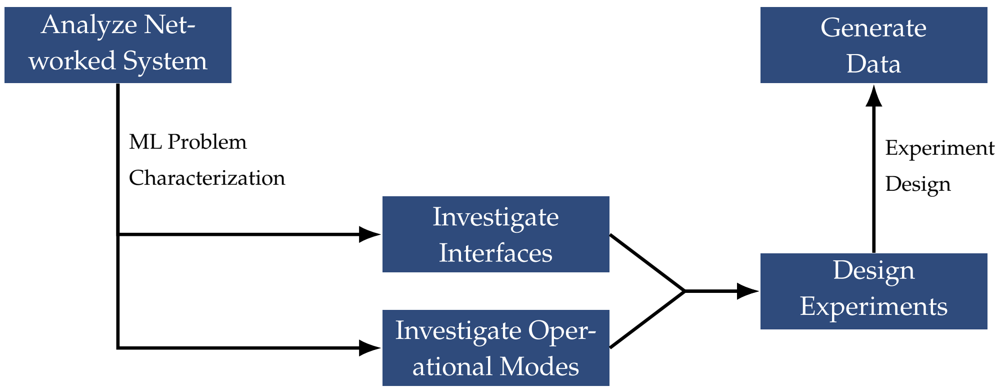


Figure 2.4 The *Analyze Networked System* sub-process investigates the available interfaces, known operational modes, and unexpected phenomena and designs experiments for the acquisition of necessary training data.

Define ML Problem. The *Define ML Problem* sub-process characterizes the ML problem using the mathematical formulation of the COP as input, and translates the network operational objective into ML objectives. The *Define ML Problem* sub-process consists of two activities: The *Characterize ML* activity and the *Formulate ML Objective* activity.

The *Characterize ML* activities decide from the COP formulation and the network operational objective whether the corresponding learning problem belongs to one of three classes: SL, RL, or UL. Further, the *ML Characterization* activity decides whether the learning problem is a classification or regression problem in the case of SL and a discrete or continuous control problem in the case of a RL problem.

Further, this step defines side conditions for the resulting ML model. For example, if the ML model should work under arbitrary inputs or inputs that are explicitly observed during network operation. For example, the ML model should serve later as a planning tool and predict performance indicators for arbitrary contingencies. Another example would be modeling the networked system's behavior during operation to detect anomalous behavior. Setting this scope is important since it directly influences the data necessary for learning and must thus be available.

With this information, the *Formulate ML Objective* can express performance requirements that the ML application should meet. For example, a specific accuracy, precision, or recall for SL problems or a specific objective value in the case of RL problems. Further, the *Formulate ML Objective* activity can define additional requirements towards the ML model, such as robustness, scalability, explainability, etc. Chapter 3 presents an example where continuous operation is important for choosing the ML model.

2.2.3 Analyze Networked System

The *Analyze Networked System* sub-process investigates the networked system for which the learning task is conducted. The *Analyze Networked System* sub-process produces experiment designs for collecting the necessary data for subsequent steps. The *Analyze Networked System* sub-process consists of three activities: *Investigate Operational Modes*, *Investigate Interfaces*, and *Design Experiments*.

Investigate Operational Modes. This activity gathers knowledge about the networked system's known phenomena and operational modes that might impact the previously defined ML objective. Phenomena are unexpected and potentially erroneous behaviors of the networked systems, whereas operational modes refer to changes in the networked system's behavior due to external influences.

Phenomena are hardware behavior, effects arising from the software implementation of algorithms and services, workload-specific behaviors, etc. For example, switches behave differently than specified [75], packet processing latency can depend on program structures [76], and links with low utilization can still drop packets [77]. The findings can be documented, e.g., with cause-effect diagrams. Phenomena that are known but unclear should also be documented. Those might be candidates for further investigation.

Operational modes of the networked system result in different behaviors of the target variables. For example, two trivial operation modes are over- and under-utilization of the networked system. In both operational modes, the specified target variable will behave differently. Depending on the system, multiple operational modes might exist.

Knowing operational phenomena and modes is important to identify additional parameters that must be observed and included in the ML model. Including those parameters is important since they affect the target variables that ML should predict. Those parameters are often not intuitive and hard to discern directly from the COP formulation. If the parameters are not included in the input space of the ML model, then the ML model cannot learn their effect on the target variables, resulting in poor performance. Knowing the causes of phenomena and the contingencies of operational modes is also necessary for designing experiments to generate data.

Investigate Interfaces. This activity analyzes the available interfaces and investigates the parameters that can be observed through these interfaces. Important to include is the cost and temporal granularity at which parameters can be observed through those interfaces. For example, Simple Network Management Protocol (SNMP) polls provide information at the granularity of seconds, whereas CPU utilization on the Operating System (OS)-level can be measured at millisecond granularity. Investigating the availability of interfaces is also important concerning the operational phase. After all, the data ML models rely on must be available during operation.

Design Experiments. This activity designs experiments, i.e., specifies measurements to obtain training data and how to store the data. This activity compares the required and the available parameters to each other. If required parameters are not directly available at the required granularity, this activity investigates if the unobserved parameters can be estimated from observable ones. If that is infeasible, the activity investigates the effort to extend the existing interfaces to include measurements of the unobserved parameters. This should also include an estimate of the measurement's effect on the networked system, i.e., if the additional overhead might interfere with the operation and thus bias the measurements.

Further, this activity specifies input to the networked system that triggers the previously identified phenomena and operational modes. Examples are traffic patterns, events in the networked system such as link and node failures, packet sequences, and mixes of traffic types.

At this point, the scope of what the ML system should learn becomes important. For example, to model the behavior of the networked system during operation, it is enough to identify and include the operational modes and phenomena that occur during operation. If the ML model should later work for arbitrary inputs, then operational modes and phenomena that might not occur during normal operation must be included in the experimental designs. This is especially important for the increasingly popular data-driven digital twins that use ML to model the behavior of networked systems. The twins are often used for planning and configuration tasks. To avoid bad configurations, the internal ML models must cover a large variety of behaviors, even if some of those behaviors are highly unlikely to occur in practice.

At the end of the *Analyze Networked System* sub-process, it might become apparent that the original project goal cannot be achieved. For example, necessary parameters cannot be observed, too many unknowns could result in project failure and should be clarified first, or the original objectives might have to be adjusted. Thus, the MaLANe process has a transition from the *Analyze Networked System* to the *Translate Objective* activity to allow the adjustment and re-evaluation of the original business objective in light of the new findings.

2.2.4 Generate Data

The *Generate Data* sub-process shown in Fig. 2.5 prepares and conducts the previously defined experiments. The sub-process produces measurements for further analysis. An important aspect of this sub-process is the level of automation, especially in conducting experiments. Ideally, experiments can be conducted automatically based on a description of the experiment in a suitable form. The sub-process consists of two activities: 1) The *Setup Experiment* activity and the *Conduct Experiment* activity.

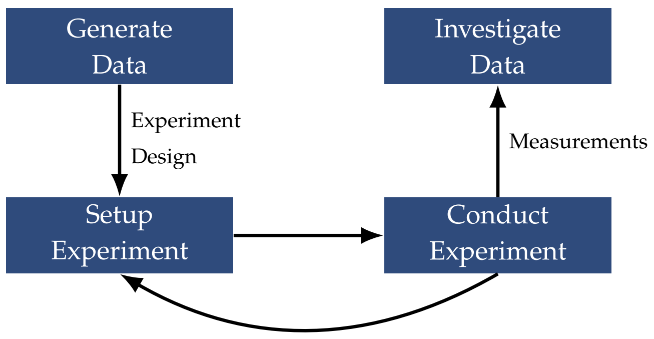


Figure 2.5 The *Generate Data* sub-process consists of the setup of a testbed in which to conduct measurements and the conduction of the previously defined experiments in the testbed.

Setup Experiments. This activity implements the necessary infrastructure to conduct the experiments. This could range from setting up measurements through established interfaces in a networked system to building a testbed. For example, listening passively to events and logs generated during normal operation might be enough. In other cases, the necessary data can only be obtained through invasive and customized hardware and software in a dedicated testbed. Further, this activity implements new interfaces to access previously identified parameters that cannot be obtained through existing interfaces. For example, fine-grained data plane statistics might require the implementation of In Network Telemetry (INT), which requires specialized hardware.

An important aspect of the experiment setup is the automation of as many steps as possible. For example, implementing the testbed such that starting experiments only requires a configuration file, i.e., no manual adjustments of system or hardware parameters are necessary. Automation is important for two reasons: 1) To gather data with minimal overhead and to make measurements reproducible. Often, multiple experiment configurations must be measured, and measurements must be repeated. Automating the measurements allows batching experiments, i.e., scheduling them once and using the time until they are finished, e.g., to investigate the already generated data. 2) the configurations can be stored along with the data, documenting the settings that produced the data. This can prove invaluable if experiments must be repeated.

If some settings cannot be automated, e.g., wiring of hosts and switches, the configuration must be recorded for the measurements obtained with the resulting configuration. Recording the configuration enables reproducibility and the search for causes if the system shows unexpected behavior.

For example, to learn the behavior of the CNF platform in Chapter 5, the existing monitoring interfaces were extended to gather CPU utilization information and network statistics at the necessary granularity. Similarly, to investigate the performance of the ML-enabled TE mechanism in Chapter 4, interfaces allowing the collection of network statistics at high frequencies had to be identified and made accessible to the prototype.

Conduct Experiments. This activity conducts the previously defined experiments in the prepared environment and stores the results as defined previously. Note that the conduction of experiments might be necessary even if only the behavior of the networked system during operation should be modeled. Certain operational modes or phenomena might rarely occur during normal operation. To obtain sufficient data for training, those phenomena and operational modes might have to be triggered explicitly. For example, the behavior of the networked system during peak events can be important but rarely occurs during normal operation. Therefore, gathering sufficient data for peak events might take a long time and potentially produce large amounts of redundant data for non-peak times.

2.2.5 Investigate Data

The *Investigate Data* activity is related to the *Data Understanding* activity in the CRISP-DM process. The *Investigate Data* activity evaluates if the networked system shows expected behaviors. For example, if the known phenomena occurred and the operational modes were triggered. Further, the *Investigate Data* activity performs an exploratory data analysis to find previously unknown phenomena and their causes. In the case of new findings or unexpected behavior, the MaLANe process adds a transition to the *Analyze Networked System* activity. The cause for the phenomena should be clarified. At least, the experiment designs should be extended to include the additional phenomena. Experience shows that projects usually require multiple iterations until a thorough understanding of the system and its behavior is obtained. The findings might even indicate that changes to the operation of the networked system are necessary to allow the successful application of ML [78]. At the end of this activity, a comprehensive dataset containing the necessary data to train ML models exists.

2.2.6 Prepare Training

The *Prepare Training* activity resembles the CRISP-DM's *Data Preparation* activity. The *Prepare Training* activity uses the comprehensive dataset from the *Investigate Data* activity to prepare the ML model training. The nature of the preparation depends on the type of ML the project uses, i.e., is different for SL, RL, and UL. This reflects in the name, which is kept more general compared to CRISP-DM's *Data Preparation*.

Supervised and Unsupervised Learning. For both ML types, data transformations such as standardization, normalization, and feature engineering, i.e., deriving new features from existing ones, might be performed. For SL and UL, the *Prepare Training* activity closely resembles CRISP-DM's *Data Preparation* activity.

In the case of SL, additional tasks such as addressing skewed classes through resampling, balancing, or sample weighting might be necessary [27, 79]. Further, for SL, the activity divides the data into a training, validation, and test set. A general rule of thumb is to use 80 % of the data for training, and 20 % for testing [27]. For hyperparameter tuning, a separate validation set might be necessary. Again, 20 % of the training set (i.e., 16 % of the full dataset) are a common choice [27]. However, depending on the problem, the learning algorithm, and the amount of data, cross-validation might be considered instead of a separate hold-out dataset for validation [27, 79].

Reinforcement Learning. Preparing training for RL is fundamentally different. Since RL uses data gathered in an environment, the *Prepare Training* activity for RL includes the design and implementation of the environment, reward function, action space, and observation space.

The environment's design and implementation rely on the data and information from the *Formalize Problem*, *Analyze Networked System*, *Generate Data*, and *Investigate Data* activity. Environment transitions, i.e., how the environment responds to an agent's actions, can be based on this information. For example, the gathered data could be used for data-driven causal models of the system [80]. Further, environment parameters, e.g., in a network simulator, can be configured based on settings and measurements in the physical system. For example, the bandwidth of links or protocol versions and setting link delays or resource consumption of applications to measured values. Finally, measured data can serve as input to drive the simulation, e.g., through empirical request patterns.

The observation and action space design also depends on the *Analyze Networked System* and *Generate Data* activity. The observation space might contain transformed and engineered features like for SL and UL. Similarly, the action space might not directly translate into a control action in the networked system. For example, a discrete action could map to a complex system configuration.

The reward function's design further depends on the *Formalize Problem* output. The reward function must reflect the network operational objective. Often the reward function is based on the objective function of a COP. Generated data informs about potential thresholds on which the reward function value depends or directly translates into reward values. For example, the reward function for load balancing in Chapter 5 directly translates to the COP objective of minimizing the maximum load. However, once a condition is fulfilled, i.e., a CPU core overloaded, the reward function returns a strong penalty.

2.2.7 Train Model

The *Train Model* activity is similar to the *Training* activity in the CRISP-DM process model. This activity selects ML models based on the characterization of the problem and trains and evaluates the models. This step does also perform hyperparameter tuning, if necessary. In contrast to the CRISP-DM process, the MaLANe process includes the model selection into this stage, i.e., the best-performing model based on the ML objective is already selected in this activity.

For the selection of ML models, additional requirements expressed during the *Formalize Problem* can influence the models considered in the first place. For example, performance requirements might restrict the size of the models. Known restrictions in inference hardware might exclude model classes such as NNs from the list of potential candidate models.

The MaLANe includes a transition from the *Train Model* activity back to the *Prepare Data* activity. This allows adjustments based on the training results. For example, the transformation of features can be changed based on the training results, or other features be selected. Experience shows that especially projects relying on RL require multiple iterations between the *Prepare Training* and the *Train Model* activity until a model is trained successfully. Especially the reward function is a common source of problems and usually requires multiple trials until RL learns successfully [81]. Common problems are sparse rewards and credit assignment problems [81]. For example, the RL problem in Chapter 5 could also be cast as a single-agent problem. However, the single agent would have to perform many successive actions, receiving reward only once all CNFs are assigned to cores or overloading a core in-between. However, the problematic assignment might not necessarily be the assignment that finally overloaded the core. Instead, an earlier assignment might be unlucky and should have been done differently. Figuring out which action should be changed to improve the reward is difficult for the agent in this setting. The reward signal does not provide feedback for it. In the multi-agent approach, Chapter 5 proposes, each agent gets immediate feedback on its decision, which improves the learning dynamics.

2.2.8 Integrate Model

The *Integrate Model* sub-process shown in Fig. 2.6 takes a trained model and returns a system architecture for the resulting ML-enabled networked system. The sub-process consists of two activities: The *Implement Model* activity and the *Evaluate ML-enabled System* activity.

Implement Model. The *Implement Model* activity realizes a test-bed or Proof-of-Concept (PoC) implementation of the ML-enabled system. Doing so can entail a significant implementation effort if the trained ML model cannot operate in the networked system directly. This can be the case if, e.g., necessary libraries

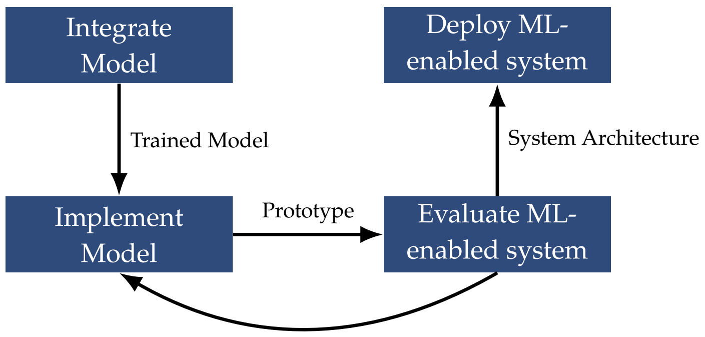


Figure 2.6 The *Integrate Model* sub-process implements the trained model in the networked system. It evaluates if the resulting ML-enabled system meets the specified network operational objectives.

are not available, the ML model must be ported to networking hardware or optimized for performance. For example, NNs are usually trained with deep learning libraries like TensorFlow or Torch. Those libraries might not be available in the networked system or add too much overhead to the resulting program, potentially necessitating the manual implementation of forward passes. Similarly, decision trees can be executed in the data plane but must be translated into match-action rules in switching chips first [56]. Other algorithms might have to be implemented in another programming language to be able to execute them in the networked system. To improve performance, optimization tricks, e.g., optimizing the memory layout to improve cache utilization, or exploiting other model-specific properties that result in performance improvements, might be necessary.

For example, Chapter 4 demonstrates that using a deep learning library can result in significant overhead, severely hurting the overall system's performance. A manual implementation was necessary to remove the overhead, and exploiting the model structure was necessary to make the system practically applicable.

Evaluate ML-enabled System. The *Evaluate ML-enabled System* evaluates the performance of the ML-enabled system the *Implement Model* results in to assure its quality or evaluate its potential. Goal-oriented projects often focus on quality assurance, i.e., verifying if the system design meets performance requirements specified during the *Translate Objective* and *Formalize Problem* activities. Exploration-oriented projects are usually more interested in the system's potential, i.e., to establish what performance could be achieved with such a system in the first place and identify limiting factors that could be turned into a business case.

Evaluating the system requires the design and execution of experiments to obtain the required measurements. In the case of exploration-oriented projects, it is recommended to follow standardized setups, e.g., defined in the RFC2544 [82]. Goal-oriented projects can require less standardized experiments, e.g., to evaluate Key Performance Indicators (KPIs) in specific contingencies. For example, to evaluate the ML-enabled TE mechanism, special experiments were designed to

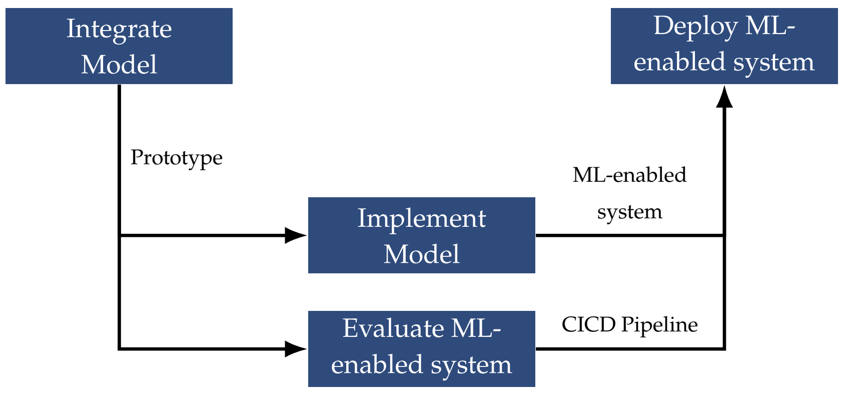


Figure 2.7 The *Deploy ML-enabled System* sub-process consists of two parallel activities: Deploying the ML-enabled system into production and automating ML related activities to allow the adaptation, e.g., to data drift.

measure how fast the system reacts to changes in the network. Similarly, different workloads were used in Chapter 3 to investigate how the workloads impact the system's performance.

Depending on the evaluation results, the *Integrate Model* sub-process specifies the return to the *Implement Model* activity. This allows the implementation improvement, for example, if the evaluation reveals bottlenecks that should be resolved, as in Chapter 4.

Further, MaLANe specifies a transition to the *Translate Objective* activity and a transition to the *Train Model* activity. Transitioning to the *Translate Objective* activity might be necessary if the *Integrate Model* sub-process shows that the resulting system cannot meet the desired objectives. Thus, the objectives might have to be adjusted accordingly. Similarly, insights gained during this process might allow the development of new business cases. Moving back to the *Train Model* activity might be necessary if the trained ML model must be adapted to meet performance requirements. For example, the model inference is not fast enough, and the model size should be reduced, or previously set ML KPIs must be adapted to achieve the desired results in the ML-enabled system, i.e., using a smaller model that might have a lower accuracy but faster inference times.

2.2.9 Deploy ML-enabled system

The *Deploy ML-enabled System* sub-process shown in Fig. 2.7 deploys the ML-enabled system into production and implements an automated Continuous Integration Continuous Delivery (CICD) pipeline. Accordingly, the *Deploy ML-enabled System* sub-process consists of two parallel activities: The *Deploy in Production* activity and the *Automate Pipeline* activity.

Deploy in Production. The *Deploy and Operate ML-enabled System* activity deploys the solution into the production environment. Best practices such as incremental deployment, Canary testing, and A/B testing apply. The deployment of the system can further include the training of personnel.

Automate Pipeline. In practice, the data distribution is non-stationary [70]. That is, the data distribution slowly changes over time. This can hurt the model performance and thus the ML-enabled system. Successfully operating a ML-enabled system thus might entail setting up and automating a CICD pipeline to keep performance high. This includes setting up monitoring to detect changes in the data distribution and model performance degradation and methods to automate the process of generating data, training the model, and deploying the updated model to the system. For this activity, the term Machine Learning Operations (MLOps) has been coined over the recent years, and various tools exist to assist this activity [83].

2.2.10 Operate ML-enabled system

The *Operate ML-enabled system* activity ensures the correct operation of the ML-enabled system. The activity uses the monitoring system to detect if the deployed model has gone stale. Further, hardware degradation and system updates might also affect the performance of the ML-enabled system. The ML component should thus be included in maintenance processes, i.e., verifying the impact of maintenance and the assumptions on which the ML model is based. Before initiating a model update, the cost of training a new model to the cost resulting from erroneous predictions should be compared. The arrow from the *Operate ML-enabled System* to the *Analyze Networked System* sub-process in Fig. 2.2 indicates this.

2.3 Conclusion and discussion

This chapter presents the MaLANe process model for conducting data science projects in networking. MaLANe adapts the popular CRISP-DM process model to the specific needs of data science projects in networking. In contrast to extensions such as CRISP-ML(Q), and DMME, MaLANe includes a perspective on combinatorial optimization, RL, and the integration of ML models into the systems.

A data science project in networking might not require the execution of all MaLANe activities. For example, the formulation of a COP might not be necessary if the goal is to model one specific aspect or component in the network, i.e., there is no underlying optimization problem to solve. For instance, to predict the CPU utilization of an application or to perform predictive maintenance on hardware components. In those cases, proposals like the CRISP-DM, CRISP-ML(Q), and the DMME process model are equally applicable.

The MaLANe process also emphasizes that ML projects can become extensive and that expertise in different disciplines is usually needed for a successful project. The MaLANe process shows that thorough technical understanding is required to successfully do ML projects in the networking domain. Experience shows that a successful ML project might even require a deeper understanding of the networked system than for normal operation. A finding that is supported by others [78].

Also, the MaLANe process gives a negative answer to whether ML can be used as a black-box method to learn the behavior in networked systems. The MaLANe process shows that the resulting ML models will have little credibility, i.e., the reliability of the predictions is unclear. Assuming that the networked system is treated as a black box, i.e., ML gets input and output data of the system, and the internal workings of the system are largely unknown. Further, assuming that the ML model meets ML and network level performance requirements, e.g., has high accuracy. In this case, the credibility of the ML system depends solely on the quality of the training data and the scenarios in which the system is evaluated. However, to judge the quality of the training data and the suitability of the evaluation scenarios, a deep technical understanding of the system and its workings is necessary, which contradicts the black-box assumption.

Further, the MaLANe process model shows that significant effort is also necessary during the operational phase. Neglecting the effort in the operational phase can quickly lead to situations where the model is no longer usable.

Chapter 3

Data-driven website classification Algorithm Design

Encrypt every transfer [84] is a major step towards securing the privacy of Internet users: DNS[1] over TLS[2] (DoT.) [85], and DNS over HTTPS[3] (DoH) [86] secure the user's DNS queries; HTTPS with the Encrypted Server Name Indication (ESNI) extension [87], and the proposed Encrypted Client Hello (ECH) extension [88], secure the communication between clients and remote servers. Hence, adversaries, legitimate or not, can no longer analyze DNS traffic, packet payload, the Server Name Indication (SNI), or the information in the TLS Application-Layer Protocol Negotiation (ALPN) extension [89]. In 2019, at least 10 % of the Alexa top 1 Million websites supported ESNI and would have been hidden [90].

For SPs, the secured communication of Internet users has a downside: They become oblivious to the applications and services running over their network. However, knowledge about applications and services is important for SPs, especially as competition strengthens. For example, SPs can use the knowledge about the service and application distribution to plan network infrastructure to improve the Quality of Service (QoS) and Quality of Experience (QoE) of users [25] and to develop new business and pricing models [2, 3, 9, 25].

Consequently, legitimate and illegitimate adversaries must rely on other methods to analyze users' communication. Website Fingerprinting (WFP) [91] might thus become relevant beyond anonymity networks such as the The onion router (Tor) network. WFP allows adversaries to infer visited websites from encrypted traffic using patterns in the data exchange between client and server.

However, using WFP at scale in the network is challenging. WFP should be deployed at a location that allows the monitoring of multiple users, i.e., WFP must operate live in the network at higher link rates (say 10 Gbit/s) on a traffic aggregate. Previous WFP attacks [91–117] are not applicable in this scenario. Previous attacks are evaluated offline, target individual users, and use ML models that

[1]Domain Name Service (DNS).
[2]Transport Layer Security (TLS).
[3]Hypter Text Transfer Protocol Secure (HTTPS).

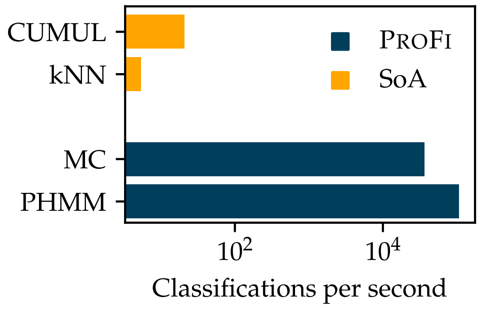


Figure 3.1 Classifications per second that ProFi and SoA approaches achieve in an offline setting on one CPU core.

take milliseconds to evaluate and thus do not scale to high-volume links. Further, most previous attacks use features calculated from entire page loads, i.e., require extracting individual page loads from the traffic aggregate. Extracting page loads is difficult considering that simply detecting the onset of one concurrent page load in a single user scenario is already challenging [101, 104, 111, 118]. Thus, it remains unclear if WFP can be used in the context of SPs.

To show that WFP applies in a general setting, this chapter designs, implements, realizes, and evaluates an ML-enabled traffic classification system called ProFi (PRObabilistic website FIngerprinting). In contrast to previous work, ProFi's design includes three operational requirements beyond mere ML metrics like detection rate: Data availability during operation, inference speed, and continuous operation. ProFi uses Probabilistic Graphical Models (PGMs)[4] [80] to model the initial TLS connection established during webpage retrieval based on the direction, size, and TLS records among the connection's first 30 packets, which are, as this chapter shows in a prototypical implementation, readily available in practice. That is, ProFi operates on individual flows instead of individual page loads, as most previous attacks do. Thus, ProFi handles traffic aggregates without further processing because of the flow-based classification. For ML-related metrics, ProFi shows competitive performance to three SoA WFP attacks. ProFi achieves precision and recall scores of 86.51 % and 85.35 % in a closed-world, and 68.90 % and 78.71 % in an open-world scenario. At the same time, the PGMs have few parameters, enabling inference within microseconds. Thus, ProFi processes thousands of webpage calls per second in an offline scenario. Two orders of magnitude more than SoA approaches, as Fig. 3.1 shows. The PGMs further simplify the continuous operation of the attack since the PGMs readily indicate data drift through their log-likelihood. Other ML models require more involved monitoring, e.g., by observing and evaluating samples during production [70]. Further, the PGMs enable a modular microservice-based implementation amenable to deployment in modern cloud-based networks such as on a Multi-access Edge Computing (MEC) [7] infrastructure. This chapter evaluates the performance of

[4]The term PGM in this chapter refers explicitly to Markov Chains (MCs) and Profile Hidden Markov Models (PHMMs).

the microservice-based system in testbed measurements, showing that ProFi's microservice-based implementation can handle more than 400 webpage accesses per second on a 10 G link at full rate. Overall, the contributions of the work in this chapter are:

- Collection of an extensive traffic dataset from 4 800 webpages of 96 websites collected over 70 days.
- Implementation of a TLS dissector for feature extraction.
- Design, implementation, and evaluation of ProFi, a PGM-based WFP attack designed for the operation live in the network.
- A microservice-based testbed implementation and evaluation of ProFi.
- Showing that ProFi can handle 100s of website accesses per second. This is the first implementation and evaluation of a WFP attack's potential to operate live in the network and has relevance beyond WFP, i.e., extends to ML-based traffic classification in general.
- The design of a readily-implementable defense against ProFi reducing precision and recall to less than 10 % and 20 % while causing a bandwidth overhead of 150 %.
- The release of data [119] and code [5] to foster reproducibility and support the community in the research of traffic classification.

Content and outline of this chapter. To improve the understanding and readability of the chapter, the chapter's sections do not directly correspond to steps in the MaLANe process model. Instead, the chapter is organized as follows. Sec. 3.1 introduces the attack scenario, and Sec. 3.2 introduces background information and related work. Sec. 3.3 presents the design of the ProFi attack and the defense method. Sec. 3.5 introduces the dataset and analyzes the training process. Sec. 3.6 evaluates the classification performance of the ProFi attack and compares ProFi to three baselines. Sec. 3.7 presents the microservice-based implementation of ProFi and evaluates its throughput, scalability, and labeling speed. Sec. 3.8 discusses ethical considerations, and Sec. 3.9 concludes this chapter. This chapter is largely based on a previously published article [31]. This chapter adds additional contents in Sec. 3.2.1, Sec. 3.2.3, and extends the description of attacks and defense in Sec. 3.3. Specifically, this chapter adds Sec. 3.3 visualizations of trained PGMs to emphasize the model's simplicity and derives the adapted training algorithm. Finally, this chapter describes the individual services comprising the prototype in more detail in Sec. 3.7.1.

[5] Available on GitHub `https://github.com/tum-lkn/ProFi-Efficient-probabilistic-traffic-fingerprinting`

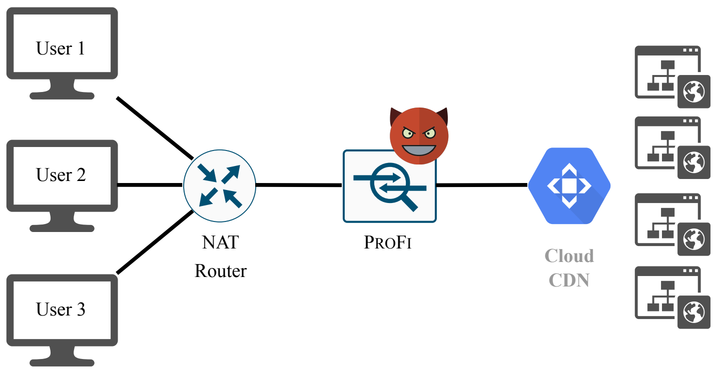


Figure 3.2 ProFi attack scenario. Multiple users access websites on a Content Delivery Network (CDN) and access the Internet via a Network Address Translation (NAT) router. ProFi sits between the NAT router and the CDN, observing only the traffic aggregate.

3.1 Attack Scenario

This section introduces the underlying use-case for MaLANe's *Translate Objective* activity and deduces network-level objectives. Further, this section characterizes additional requirements towards the ML models as described for the *Characterize ML* activity. Fig. 3.2 illustrates the attack scenario of ProFi: Multiple users share a common Internet access point- a typical Internet access scenario (e.g., home network or public WiFi).

The user view. The users are situated behind a NAT-capable router, as in a typical home or campus network, which renders a realistic attack scenario. The users share the same public Internet Protocol (IP) address; hence, they cannot be differentiated by their IP addresses. Since most of today's web traffic is carried by large Content Delivery Networks (CDNs) [120–122], this chapter focuses on a scenario where users retrieve webpages from CDNs. This chapter focuses on CDNs since CDNs serve many popular websites [112], and websites can use CDNs to hide from censors [123]. Since a CDN server can deliver any website hosted by the CDN, WFP becomes more challenging since the server's IP address does not identify one website. This chapter does not investigate the effect of Virtual Private Network (VPN) on WFP. Over two-thirds of Internet users do not use a VPN [124].

The attacker view. The goal of the attacker is to discern the website a user is accessing, where a website consists of multiple webpages. The attacker is situated on the path between the NAT router and the CDN. Contrary to previous WFP attacks, the attacker has no access to the private network of the users. However, the attacker still has access to traffic in both directions. It is assumed that the traffic is encrypted with TLS, and the SNI and ALPN extensions in the TLS handshake are also encrypted. This prevents usage of either feature for fingerprinting [125]. Further, the attacker knows only the public IP of the NAT router for every user. Furthermore, as websites are hosted on CDNs , attackers cannot fingerprint websites based on the IP addresses of destination (cache or edge) servers. Thus, this chapter investigates a scenario where the attacker is left with statistical information about the frequency, volume, and direction of data transfers within a flow.

Per-Flow Classification. ProFi classifies based on the first TLS connection the user's client establishes to retrieve a webpage. ProFi uses observable features of the traffic, such as the frame size, TLS record size, and the direction of packets. The underlying assumption that this chapter confirms in the evaluation is that the server and client exchange unique information in the initial flow, which reflects in a specific pattern of exchanged packets in the network. Thus, the attack can classify each flow individually, sidestepping the extraction of individual webpage calls from the traffic aggregates, as previous attacks require, which is very challenging to perform in practice. Previous work shows that even solely identifying the *start* of overlap for two overlapping webpage calls of a single user is difficult [101, 104, 111, 118].

Practical feasibility. Since this chapter investigates whether WFP applies in a general setting, the attack must be able to process larger traffic volumes, requiring fast and efficient WFP mechanisms. Further, the attack must be deployable and continuously operational, i.e., this chapter considers not only training and evaluation but also the full life cycle of an ML application [70]. Therefore, this chapter does not use more complex models like Random Forests, Support Vector Machines (SVMs), and Deep Neural Networks (DNNs)[6]. Specifically, there is a trend of using DNNs for WFP [105, 106, 109, 111, 116]. However, DNNs are large, with many layers and parameters. For example, Var-CNN [109] uses two Convolutional Neural Network (CNN) with 18 layers and an input layer with 5 000 neurons each. Despite their size, simpler ML models can outperform DNNs on traffic classification tasks [126]. Further, executing DNNs requires specialized hardware and a specifically designed system architecture to operate online in the network [23]. Further, DNNs are susceptible to shortcut learning, and results are difficult to verify [127]. Thus, this chapter focuses on simple ML models that run on commodity hardware and are easy to interpret. This makes the development and deployment of WFP attacks faster since models require less computation, and also helps in interpreting and analyzing the learned models, e.g., to mitigate the WFP attack.

[6]Random Forests and SVMs can also yield large models that take milliseconds to evaluate a sample and are thus unsuitable for WFP attacks at scale. For example, CUMUL in Fig. 3.1 is based on an SVM.

Code	Record Type	Code	Handshake Type
20	Change Cipher Spec		
21	Alert		
22	Handshake	0	`hello_request`
		1	`client_hello`
		2	`server_hello`
		3	`hello_verify_request`
		4	`new_session_ticket`
		5	`end_of_early_data`
		6	`hello_retry_request`
		8	`encrypted_extensions`
		11	`certificate`
		12	`server_key_exchange`
		13	`certificate_request`
		14	`server_hello_done`
		15	`certificate_verify`
		16	`client_key_exchange`
		20	`finished`
		21	`certificate_url`
		22	`certificate_status`
		23	`supplemental_data`
		24	`key_update`
		254	`message_hash`
23	Application Data		

Table 3.1 TLS record- and handshake message types.

3.2 Background and related work

This section introduces background information and related work and reflects MaLANe's *Analyze Networked System* activity. Sec 3.2.1 introduces background information for the TLS protocol. Sec. 3.2.2 introduces WFP in detail, and Sec. 3.2.3 introduces background information to Hidden Markov Models (HMMs). Finally, Sec. 3.2.4 discusses related work.

3.2.1 Transport Layer Security

TLS provides a secure channel between two communication partners [128]. Currently, TLS uses mostly the Transmission Control Protocol (TCP) for transmission. However, Quick UDP Internet Connections (QUIC) is gaining traction. This chapter only considers TLS over TCP.

TLS consists of two major components: A handshake and a record protocol. The handshake protocol authenticates the peers, establishes keying material and negotiates cryptographic techniques and their parameters [128]. The record protocol fragments data into records and encrypts the content of the records using the settings established during the handshake [128]. In addition, TLS has an

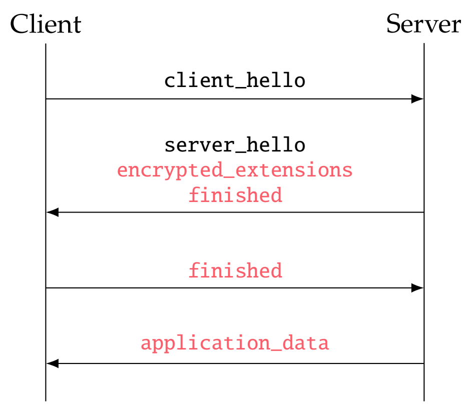


Figure 3.3 TLS connection establishment. Red indicates encrypted messages.

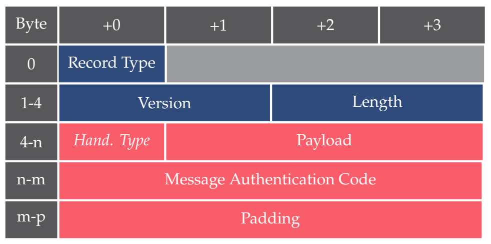


Figure 3.4 Structure of a TLS record. Blue color indicates clear text, red color indicates encrypted content. Italic font indicates content only present in specific messages.

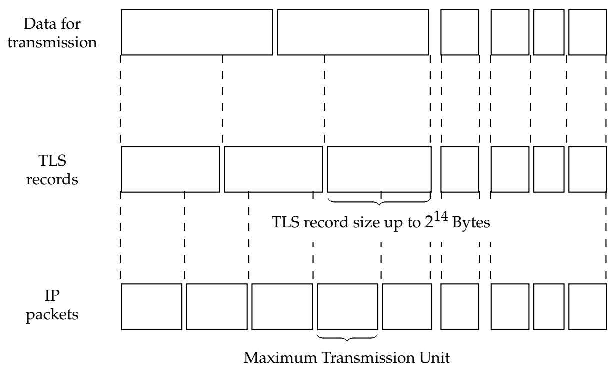

Data for transmission
TLS records
TLS record size up to 2^{14} Bytes
IP packets
Maximum Transmission Unit

Figure 3.5 Data is first fragmented into TLS Records. The TLS Records are then fragmented into IP packets.

Alert protocol that is used to communicate errors and initiate the closure of the connection [128].

Fig. 3.3 shows how a TLS session is initiated. For brevity, the TCP handshake and control messages are omitted. A client initiates a TLS session by sending a `client_hello` to a server. The message is part of the handshake protocol and contains client random data, a list of supported cipher suites, a list of public keys, and protocol versions supported by the client. In addition, the `client_hello` can contain several optional messages [128]. The `client_hello` is sent as plain text, i.e., nothing is encrypted. Upon receiving a `client_hello`, the server replies with a `server_hello`. The `server_hello` contains server random data, a selected cipher suite, a public key for key exchange, and the negotiated protocol version. The `server_hello` is the last unencrypted message in the connection [128]. All subsequent handshake and application data messages are encrypted. The client finishes the connection establishment by sending a `finished` message to the server. The client and server can then send application data in both directions. Depending on the configuration of client and server and network conditions, additional messages can be exchanged during the TLS handshake process [128]. Tbl. 3.1 lists all TLS record types and TLS handshake message types. ProFi uses the message types as features.

The content of messages exchanged during a TLS session is framed. Frames can have a size of up to 2^{14} bytes, and each frame has a header. Fig. 3.4 illustrates the header. The header has a length of five bytes and specifies the record type, the TLS version, and the length of the record. Headers of messages from the handshake protocol have a sixth byte after the length that contains the handshake type. The first five to six bytes are *never* encrypted. After the header, the actual payload of the TLS record begins. At the end of the record are a Message Authentication Code (MAuC) with four bytes and arbitrary padding. TLS uses the MAuC to verify message integrity and authenticity. The padding can be used to mask the size of the payload. Payload, MAuC, and padding are encrypted. ProFi uses the visible data, i.e., the record, handshake type, and the record length, as features.

Since frames can have a length of up to 2^{14} Bytes, TLS records can be fragmented into multiple IP packets. Fig. 3.5 illustrates this process. Fig. 3.5 shows six chunks of data that should be transmitted. Each chunk of data is framed by TLS. This results in seven TLS records. The TLS records are then passed to the transport protocol. In this example, the Maximum Transmission Unit (MTU) in the network is smaller than the maximum record size. The first three TLS records that have maximum size are therefore fragmented into five IP packets. The fourth TLS record is packed into a single packet. The last three TLS records are packed into one packet. How records are created and sent into the network depends on the application [128]. ProFi uses the number and types of records inside packets as a feature.

3.2.2 Website Fingerprinting

The goal of a WFP attack is to predict which websites a user is accessing solely from passive observation of network traffic [94]. In this context, it is important to differentiate between web-*sites* and web-*pages* [95].

Website vs. webpage. Attacks can infer web*sites* and web*pages* [95]. A web*page* is identified by a specific Unified Resource Locator (URL). A web*site* consists of multiple webpages below a Second Level Domain (SLD). For example, `https://www.nih.gov/grants-funding` and `https://www.nih.gov/research-training` are two webpages from one website. Related work often performs WFP on the level of webpages, e.g., the landing page—here, the attacker has to learn a model that recognizes only this single page. This task is easier than learning a model for arbitrary webpages of one website. A website's webpages show different content, resulting in different packet sequences. Thus, website fingerprinting has to cope with a larger variance in the data compared to webpage fingerprinting [95], making the data more difficult to model [27, 129]. ProFi performs the more challenging website fingerprinting attack, i.e., ProFi extracts patterns that generalize from examples of webpages of websites to the traffic of unseen webpages of that site.

Open vs. closed world. Two evaluation scenarios exist for WFP attacks: the closed-world and the open-world scenario [94]. In the closed-world scenario, the attacker knows the websites a victim visits. Thus, the attacker can train the attack model for each of those websites - no other websites are visited. This is the easiest variant of the attack. In the more realistic open-world scenario, the attacker does not know which websites the victim accesses. The victim also visits websites not included in the attacker's training data. Generally, the open-world scenario is considered the more challenging scenario. This chapter shows the results for both scenarios.

3.2.3 Hidden Markov Models

This section introduces the HMM and its parameter estimation. ProFi uses a special HMM type to model sequences. The section is based on the tutorial of Rabiner [130], Murhpy [27], and Koller [80].

Introduction

A HMM is a stochastic model for data with discrete time and states [27] and emissions. Specifically, HMMs can model sequences with discrete sequence elements, e.g., text, genomes, and speech [27, 80, 130]. In contrast to a MC, HMMs are latent-variable models capable of capturing long-range dependencies between

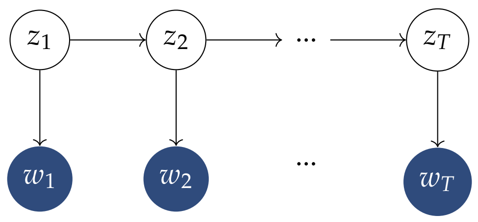


Figure 3.6 Directed graphical model of a HMM. The filled nodes at the bottom correspond to observed variables. The white nodes at the top correspond to unobserved hidden variables.

observations [27]. HMMs assume the existence of an unobserved underlying process that can be modeled by a first-order MC, and sequences are the noisy output of this process [27]. Fig. 3.6 illustrates this principle. Fig. 3.6 shows a directed graphical model of a HMM, where the unfilled nodes at the top correspond to the unobserved first-order MC, and the gray nodes at the bottom to the observed variables.

Thus, a HMM has a set of unobserved states $\mathcal{Z}$, and observable values $\mathcal{W}$. The parameters of a HMM are a transition model and an emission model [27, 80, 130]. The transition model parameterizes the unobserved first-order MC, i.e., specifies the probability of starting in any of the states of the unobserved MC:

$$\pi : \mathcal{Z} \to [0, 1], \tag{3.1}$$

and the probability of transitioning from a state $z \in \mathcal{Z}$ to another state $z' \in \mathcal{Z}$:

$$\tau : \mathcal{Z} \times \mathcal{Z} \to [0, 1], \tag{3.2}$$

referred to as $\tau\,(z_{t+1} = z' \mid z_t = z)$. The emission model defines the probabilities of observing a specific symbol $w \in \mathcal{W}$ given a hidden state $z \in \mathcal{Z}$:

$$o : \mathcal{W} \times \mathcal{Z} \to [0, 1], \tag{3.3}$$

referred to as $o\,(w_t = w \mid z_t = z)$. Estimating a HMM's parameters from observed sequences can be done with the Baum-Welch algorithm, a specific form of the Expectation Maximization algorithm [80, 130].

The Baum-Welch algorithm consists of the forward-backward algorithm and a parameter re-estimation.

Forward-Backward Algorithm

The forward algorithm uses dynamic programming to calculate the probability of an observed sequence $O \in \mathcal{W}^T$ of length T given a HMM, i.e., $p(O \mid \pi, \tau, o)$ [130]. The forward-backward algorithm uses two recursively estimated quantities: the forward variables α_t and the backward variables β_t. The forward variables are defined as:

$$\begin{aligned}
&\alpha_t : \times_t \mathcal{W} \times \mathcal{Z} \to [0,1]; \\
&\quad O_1, \ldots, O_t, z \mapsto \begin{cases} \pi(z)\, o(O_1 \mid z) & \text{if } t = 1 \\ \left[\sum_{z' \in \mathcal{Z}} \alpha_{t-1}(z')\, \tau(z \mid z')\right] o(O_t \mid z) & \text{if } 1 < t \leq T \end{cases}
\end{aligned} \tag{3.4}$$

i.e., the forward variables give the probability of observing the sequence O up to time t and being in a specific hidden state $z \in \mathcal{Z}$ at time t [130]. The explicit reference to $O_{1:t}$ is dropped from the arguments of α_t to keep the notation uncluttered. For each hidden state $z \in \mathcal{Z}$, the forward variables are initialized with the product of starting in z and observing the first symbol O_1 in z. For the remaining symbols, the corresponding forward variables are computed recursively. The interpretation of the calculation of the remaining forward variables uses the definition that each $\alpha_{t-1}(z')$ represents that the partial sequence $O_{1:t-1}$ has been observed and state z' is reached. Then, to observe symbol O_t in state z, Eq. (3.4) multiplies each α_{t-1} of each hidden state z' with the probability of transitioning from z' to z. Summing these products up results in the probability of being at state z in time t and having observed the sequence $O_{1:t-1}$. By multiplying the sum with the probability of observing O_t in state z, Eq. (3.4) computes the probability of observing the partial sequence $O_{1:t}$ and being in state z [130].

Further, the forward variables allow the computation of observing the sequence O given the HMM as [130]:

$$p(O \mid \pi, \tau, o) \doteq \sum_{a \in \mathcal{Z}} \alpha_T(a)\,. \tag{3.5}$$

The backward variables are similarly defined [130]:

$$\begin{aligned}
&\beta_t : \times_{T-t} \mathcal{W} \times \mathcal{Z} \to [0,1]; \\
&\quad O_t, \ldots, O_T, z \mapsto \begin{cases} 1 & \text{if } t = T, \\ \sum_{z' \in \mathcal{Z}} \tau(z' \mid z)\, o(O_{t+1} \mid z')\, \beta_{t+1}(z') & \text{else,} \end{cases}
\end{aligned} \tag{3.6}$$

i.e., the backward variables give the probability of the remaining observation sequence $O_{t+1:T}$, given a specific state $z \in \mathcal{Z}$ for time t. As before, the explicit reference to $O_{t:T}$ is omitted from the arguments of β_t to keep the notation uncluttered. Eq. (3.6) initializes the backward variable for each hidden state to one. In the recursive step, Eq. (3.6) accounts for all possible states for time $t+1$, the probability of observing symbol O_{t+1}, and accounting for the remaining partial observation sequence until the end of the sequence, corresponding to the β_{t+1} variables [130].

Parameter Update

Updating a HMM's parameters, i.e., τ and o, from a set of sequences $\mathcal{X}$ uses the forward and backward variables computed by the Forward-Backward algorithm [130]. Intuitively, the Baum-Welch algorithm can be interpreted as counting events, e.g., the expected number of times the HMM transitions from a state z to a state z' divided by the expected number of times the HMM is in state z. Since this thesis focuses on PHMMs and the probability π is constant for those [130], its re-estimation is omitted.

The transition probabilities τ^{i+1} are re-estimated as follows [130]:

$$\tau^{i+1}(z' \mid z) \doteq \frac{\sum_{O \in \mathcal{X}} \frac{1}{p(O|\tau^i, o^i)} \sum_{t=1}^{|O|-1} \alpha_t^O(z') \tau^i(z' \mid z) o^i(O_{t+1} \mid z') \beta_{t+1}^O(z')}{\sum_{O \in \mathcal{X}} \frac{1}{p(O|\tau^i, o^i)} \sum_{t=1}^{|O|-1} \alpha_t^O(z) \beta_t^O(z)}. \tag{3.7}$$

The superscript to τ and o indicates the iteration and the superscript on α_t and β_t the sequence from the training set, i.e., each sequence is associated with its own forward and backward variables.

The observation probabilities o^{i+1} are re-estimated as follows [130]:

$$o^{i+1}(o \mid z) \doteq \frac{\sum_{O \in \mathcal{X}} \frac{1}{p(O|\tau^i, o^i)} \sum_{t=1:O_t=o}^{|O|-1} \alpha_t^O(z) \beta_t^O(z)}{\sum_{O \in \mathcal{X}} \frac{1}{p(O|\tau^i, o^i)} \sum_{t=1}^{|O|-1} \alpha_t^O(z) \beta_t^O(z)}. \tag{3.8}$$

To obtain the final values, the Baum-Welch algorithm thus iterates between computing the forward and backward variables, updating the parameters with those, and then re-calculating the forward and backward variables with the updated distributions.

3.2.4 Related Work

This section introduces work related to the work in this chapter. Since ProFi is an interdisciplinary project touching different areas, the related work is separated into four aspects. First, this section introduces prior attacks and differentiates ProFi from them. Second, this section reviews existing defenses; third, it introduces prior art in general traffic classification. Finally, this section reviews related work in the area of real-time in-network classification systems.

WFP attacks. Many recent attacks perform web*page* fingerprinting, e.g., try to detect the landing pages of websites [91, 93, 94, 97, 98, 100, 101, 103–107, 109–116, 118, 131–133]. Fewer works perform the more challenging web*site* fingerprinting [95, 102, 108, 117]. Except for [100, 101, 104, 108, 117, 118], previous work makes the single-page-load assumption [101], i.e., users load only one webpage and there is no background traffic. All previous work evaluates their attacks under a single-user assumption, i.e., only one active user or process generates traffic.

Closest to our approach is the work of Shen et al. [110], Zhuo et al. [108], Hoang et al. [112], and Trevisan et al. [117]. Shen et al. classify normal TLS traffic from one website and use only traffic from TLS sessions with the SNI of the top-level domain, the first 100 packets of a webpage load and the kNN classifier. Zhuo et al. use a PHMM and consider subsequent webpage accesses of a single user. Hoang et al. build fingerprints from the IP addresses of webpage calls and is thus computationally efficient. Trevisan et al. [117] use per-flow features and a Random Forest classifier to label flows with domain names. In contrast, ProFi uses a novel WFP classification method based on PGMs and anomaly detection that makes ProFi easy to train and execute. The main contribution of this work is the evaluation of the attack in a testbed, i.e., the investigation if and at what scale WFP can be executed online in the network and interfere with user traffic. ProFi is different from previous WFP attacks on the Tor network in the following ways. ProFi makes predictions for flows and does not need access to all packets of each webpage call. Further, ProFi does not assume the monitoring of an individual user.

WFP defenses. A number of mechanisms to mitigate WFP attacks exists [96, 99, 113, 134–150]. The goal of the defenses is to mitigate WFP in the Tor network. To mitigate WFP, the defenses mutate various aspects of the communication to render previously identified features useless. However, mitigating WFP attacks results in bandwidth and latency overheads. This chapter presents a simple defense based on the padding of TLS records. The defense is similar in spirit to existing defense methods such as CS-BuFLO [99]. Instead of the Tor network, the defense focuses on the TLS protocol and uses only mechanisms available in the current TLS standard.

General traffic classification. Beyond WFP, statistical approaches have been used for general network traffic classification [151, 152]. These works focus on broadly identifying application classes, e.g., Peer-to-Peer (P2P), e-mail, or web, using behavioral analysis [22] or detecting Secure Shell (SSH) and Skype flows [153] or Android Apps [154], both using flow-level features. Similarly, TLS features were used to identify services (e.g., web vs. video) and applications (e.g., Facebook chat vs. Google Drive) [155] or web services [156]. Unlike application or service detection, ProFi focuses on performing WFP attacks in both an open and a closed-world scenario. However, the principle behind ProFi could be applied to more general traffic classification, which is left as future work.

Real Time Traffic Classification. FENXI [23] introduces a system that uses specialized hardware accelerators for deep learning to accelerate and enable the application of deep learning to packet processing. ILSY [56] implements four ML models in programmable data planes with the P4 language and shows the

Symbol	Definition	Explanation
$\mathcal{W}$		Set of all possible symbols.
$\mathcal{U}$		Set of possible IP packets from TLS sessions.
$\mathcal{O}$	$\mathcal{O} \subseteq \mathcal{W}$	Set of all observations of one website.
z		Hidden State of a HMM.
$\mathcal{Z}$		Set of all hidden states of a HMM.
σ	$\mathcal{U} \rightarrow 2^{\mathcal{W}}$	Function that maps a packet to a ordered set of symbols.
Classifier	$\mathcal{W} \rightarrow \mathcal{Y}$	Classifier that maps a sequence of symbols to a website.
$\mathcal{Y}$		Set of websites that should be detected.
pos	$\mathcal{U} \cup \mathcal{R} \rightarrow \mathbb{N}$	Returns the position of a packet or TLSRecord in the flow. Can be considered as an increasing counter.
$\mathcal{R}$		Set of all possible TLS Records.
recs	$\mathcal{U} \rightarrow 2^{\mathcal{R}}$	Function that returns the TLS Records of an IP packet.
$\mathcal{X}$	$\{x_1, x_2, \ldots, x_n \mid x_i \in 2^{\mathcal{W}}\}$	Training set consisting of sequences for websites.
$\mathcal{X}_y$		All sequences in the training set for website y.
$\mathcal{X}_{yj}$		The jth sequence of website y.
$\mathcal{T}$	$\{\texttt{packet}, \texttt{record}\}$	Sequence element types, either IP packets or TLS records.

Table 3.2 Definition of symbols and functions sued in this section.

limitations of these approaches. Similarly, Xavier et al. [157] investigate how ML algorithms can be executed in the data plane. N2Net [59] and BaNaNa Split [57] discuss the implementation of binarized neural networks in the data plane. Pacheco et al. [158] review ML-based solutions to network traffic classification. In contrast to previous work, this chapter does not rely on special hardware nor highly optimized frameworks targeting specific application scenarios only. Instead, this chapter shows how existing frameworks and commodity hardware can be used to realize WFP attacks.

3.3 Attack Model and Defense

This section introduces ProFi's classifier using probabilistic models of packet sequences and the associated feature engineering. That is, this chapter includes most aspects of MaLANe's *Prepare Training* and *Define ML Problem* activities. The classifier is designed for computational efficiency, i.e., it relies on models with few parameters. Further, the classifier is designed for implementation as mi-

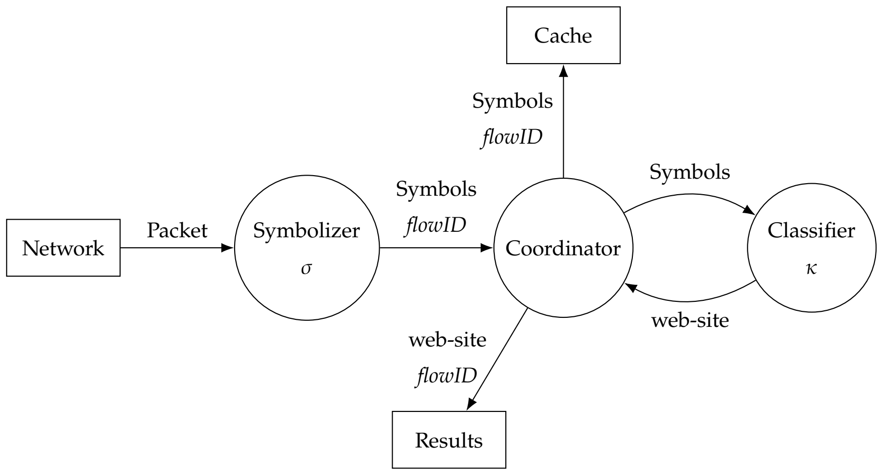


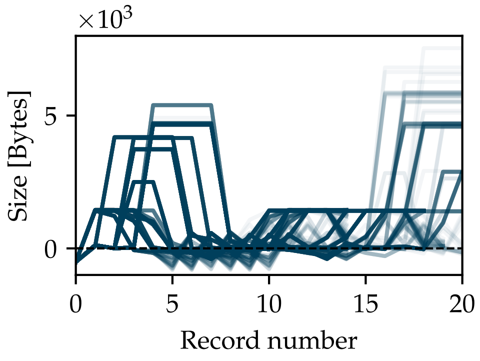
Figure 3.7 Data-flow diagram of the classification procedure of ProFi. Italic script identifies meta-data.

croservices in a cloud-native deployment scenario. The design aligns with future networking concepts that feature networking on top of a MEC infrastructure. The design assumes that ProFi is situated between the user and the CDN, routing is symmetric, and no VPN is used. Further, this section introduces three SoA approaches to WFP as baselines for later evaluations.

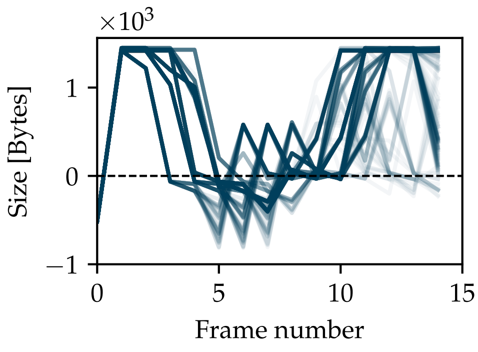
3.3.1 Overview

To operate on the flow level, ProFi learns the characteristics of the first TLS connection established between a user's client when retrieving a webpage, referred to as MainFlow from now on. Thus, classifiers in ProFi answer the question: *Is this flow a MainFlow to a webpage of one of the websites that should be detected?* Therefore, ProFi doesn't have to extract flows constituting a webpage retrieval from the traffic aggregate, i.e., the data ProFi requires is readily available during operation. Further, detecting webpage calls based on the initial connection can enable active interference, e.g., by dropping all traffic for a specific IP for a short time after the call is detected.

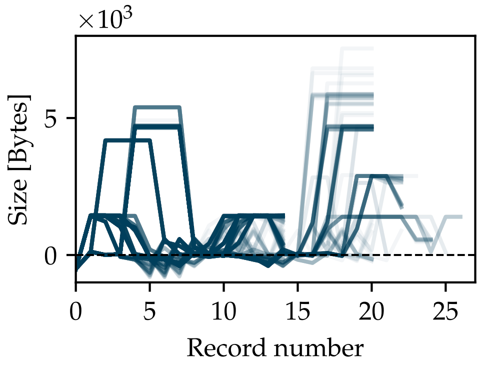
Fig. 3.7 illustrates the high-level operation of ProFi. The attack translates a flow's packets into a sequence of nominal symbols and labels the sequence with a classifier. More formally, the attack consists of three functions: The Symbolizer σ, the Coordinator, and the Classifier κ. In addition, there are three data sources and sinks, the Network, a Cache to store intermediate results, and persistent storage for the classification results. First, ProFi receives a packet $u \in \mathcal{U}$ from the network. The set $\mathcal{U}$ corresponds to all IP packets occurring during a TLS session. A Symbolizer σ converts the packet into an ordered set of abstract symbols $\mathcal{M} \subset \mathcal{W}$, where $\mathcal{W}$ is the set of all possible symbols. $\mathcal{M}$ together with a flow identifier is passed to the Coordinator. The Coordinator accumulates $\mathcal{M}_t = \sigma(u_t)$ derived from T packets $u_1, \ldots, u_T$ of the same flow. Once the T^{th} packet of a flow arrives, the Coordinator passes the accumulated sequence of



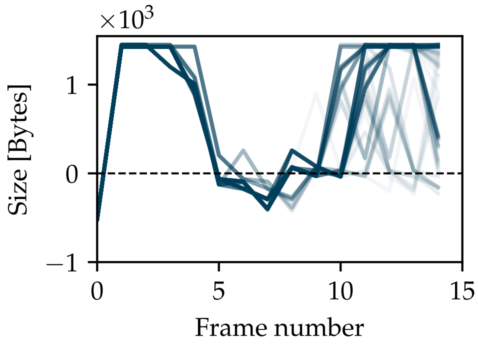
(a) TLS record sizes of `www.google.es`



(b) Frames sizes of `www.google.es`



(c) TLS record sizes of `www.primevideo.com`



(d) Frame sizes of `www.primevideo.com`

Figure 3.8 Frame-/TLS record sizes: negative (positive) values indicate traffic send by the client (server), resp.

symbols to a Classifier κ. The Classifier predicts the website $y \in \mathcal{Y}$ and returns the result to the Coordinator, where it is stored.

3.3.2 Extracted Features

To classify flows, ProFi uses packet-level data from, at most, the first 30 packets, corresponding to the 10th percentile of MainFlow lengths from the collected data. The exact number of packets used by ProFi is subject to parameter optimization of the attack.

ProFi extracts features on the level of the packet and the level of the TLS records. For each packet, ProFi extracts the size in bytes, the TLS record types, and the direction, i.e., whether the packet travels from the client to the server or the other way around. For each TLS record in the packets, ProFi extracts the TLS record length, the TLS record type, and the direction. Since existing tools such as *ssldump* do not provide all of these features, the features are extracted from the packet capture files with a custom dissector. ProFi does *not* use timing information between packets or TLS records. Latency, especially on the Internet, expresses high variance not caused by the visited website [159]. Instead, geographic location and difficult-to-observe conditions such as the network load explain the latency's variance [159]. Including latency would thus increase the requirements for data collection to avoid the risk of overfitting.

Fig. 3.8 illustrates the extracted features for two websites: `www.google.es` and `www.primevideo.com`. Negative numbers in Fig. 3.8 correspond to traffic sent from the client to the server. Positive numbers indicate the opposite direction. Fig. 3.8 shows clear differences in the constructed sequences for the websites. For example, Fig. 3.8a shows that `www.google.es` has a high variance during the TLS handshake, and the client sends larger packets to the server. In contrast, `www.primevideo.com` in Fig. 3.8c has a different handshake behavior, and the client sends less data to the server. Also, the order is important: for example, `www.google.es` in Fig. 3.8a has characteristic dips between records seven and ten. The sequence of `www.primevideo.com` in Fig. 3.8c has small records after the handshake until frame 15, after which the record sizes become highly variable. Moreover, websites can have a multi-modal behavior. For example, `www.google.es` in Fig. 3.8a has three TLS handshakes that vary in size and number of TLS records. The varying handshake lengths then shift the entire sequence. The remaining steps in ProFi, i.e., the Symbolizer and the Classifier, have to preserve and extract these patterns.

Lastly, the sequences constructed from TLS records and frames can differ. For example, the sequence of frame sizes for `www.primevideo.com` in Fig. 3.8d differs markedly from the TLS record sizes in Fig. 3.8c. TLS can package multiple records into one TCP segment. Similarly, a TLS record can span multiple TCP segments. Thus, the sequences of TLS record sizes and the frame sizes that carry the records can differ strongly. The way TLS records are packaged into TCP segments is a discriminating feature on its own [156]. In the collected data, the number of TLS records and packets is approximately equal on average but varies from website to website, confirming the potential value of this feature.

3.3.3 Symbolizer

The Symbolizer converts the features extracted for each packet and record into a sequence with nominal elements that the Classifier can process. Each sequence element is constructed from three features: direction, size of record or packet, and TLS record type(s). The size of the packet or record is numerical, and the direction and TLS record types are nominal. The Symbolizer maps the direction, size, and record types to an abstract symbol. To reduce the observation space and account for small deviations in the lengths of packets and records, the Symbolizer discretizes the packet and record lengths.

Formally, the Symbolizer is defined as a function $\sigma : \mathcal{U} \times \mathcal{T} \rightarrow 2^{\mathcal{W}}$. The set $\mathcal{T}$ corresponds to the element for which a symbol should be created, i.e., `packet` or `record`. The output is an ordered set of symbols. The order is established with the help of a function $\text{pos} : \mathcal{W} \cup \mathcal{R} \rightarrow \mathbb{N}$, returning the position of the passed element inside the flow. The symbol is formed by combining the direction, record type(s), and size.

For example, packet u travels from the server to the client and carries three application data records. The packet has a size of 1310 Bytes. The records have lengths of 510, 399 and 401 Bytes. For a $\mathcal{T}$ of `packet`, packet u would be mapped to one symbol $\{$`(23|23|23;1310;C)`$\}$, i.e., σ returns a singleton-set. In the case of $\mathcal{T}$ of `record`, packet u is mapped to three symbols. One symbol per record: $\{$`(23;620;C)`, `(23;340;C)`, `(23;350;C)`$\}$.

The input to the Classifier is the union of the return values of σ for the first n packets of a flow $\mathcal{F} := \{u_1 \dots, u_n\}$: $O := \bigcup_{u \in \mathcal{F}} \sigma(u)$. This forms a sequence with nominal elements that are sorted based on the order in which the corresponding packets occur in the flow.

3.3.4 Anomaly detection-based classifier

The Symbolizer returns sequences with nominal elements, and ProFi uses ML techniques operating on those. This chapter investigates two ML techniques: first-order MCs [156], and PHMM [160, 161].

MC and PHMM belong to the class of PGMs. ProFi uses PGMs to describe the nominal sequences obtained from the Symbolizer for webpages belonging to one website in the language of probability theory [80]. That is, one PGM represents one website $y \in \mathcal{Y}$. PGMs are chosen over other ML models because PGMs have few parameters, are interpretable, extensible, and signal data drift. A small number of parameters is important to make ProFi memory and computationally efficient, allowing ProFi to scale to many concurrent webpage accesses. At the same time, a small number of parameters aids the interpretability of the model, which is further improved through the specific structure of, e.g., the PHMMs [160]. PGMs are readily extensible, e.g., it is straightforward to include additional variables that can improve the classification [80], e.g., the information of the CDN that is contacted. Lastly, PGMs model the data directly and thus provide a measure of how well the model fits the data. This is an important aspect for the operation of ProFi since it allows the detection of data drift, i.e., the data distribution in operation changes over time compared to the distribution of the training data. An effect that has been previously reported [132] and exists in the data as well.

PGMs do not return a specific label. Instead, PGMs compute the *likelihood*, i.e., the probability $p_y(O)$ of a sequence O given the model for website y [80]. A Maximum Likelihood Classifier (MLC) can use the likelihood for classification: An MLC returns the label corresponding to the model with the largest likelihood, i.e., the model under which the observation has the highest probability [161]. However, MLCs are unsuitable for an open-world scenario. An MLC will always return a label, even if every model has a zero probability.

To overcome this issue, ProFi treats classification as anomaly detection. The classifier's PGMs represent the normal behavior of websites. If an observation shows anomalous behavior, i.e., has a low probability under the PGM describing the website's behavior, the classifier rejects the observation. Anomalous behavior is defined based on the anomaly score $\eta_y(O)$ of a website as:

$$\eta_y(O) := \frac{\log p_y(O)}{\gamma \max_{O' \in \mathcal{X}_y} \log p_y(O')}. \tag{3.9}$$

Here, $p_y(O)$ corresponds to the model of website $y \in \mathcal{Y}$, $\mathcal{X}$ corresponds to the training set, $\mathcal{X}_y$ corresponds to website y's training set, and $\gamma \in \mathbb{R}$ is a scaling factor. The score $\eta_y(O)$ corresponds to the ratio of the log-likelihood of the model for website y to the largest log-likelihood from the training set. The free parameter γ can be used to tune the ratio. The ratio's interpretation is as follows: If $\eta_y(O) \leq 1$, then the given sequence fits equally well to the model as the training data. If $\eta_y(O) > 1$, then the sequence is less likely than all training sequences.

In the closed-world scenario, the classifier returns the website corresponding to the model with the smallest score. This results in a multi-class classifier that returns a label $y \in \mathcal{Y}$ for every observation. In the open-world scenario, the classifier uses the score to turn each PGM into a binary classifier. If $\eta_y(O) > 1$, then the model rejects the observation O, i.e., observation O does not correspond to a MainFlow of website y. If $\eta_y(O) \leq 1$, then observation O is labeled as an instance of website y. This results in three cases: 1) All models reject sequence O, 2) one model accepts O, and 3) multiple models accept sequence O. In case one, the sample is classified as background traffic. In case two, the sample is classified as the website the model corresponds to. In case three, the classifier returns the website whose model has the smallest score.

Adjusting γ allows trading false positives for false negatives. A value less than one reduces the risk of false positives and increases the risk of false negatives, i.e., the classifier is stricter. A value larger one increases the risk of false positives and reduces the risk of false negatives, i.e., the classifier is more lenient in its opinion of what is normal.

This approach has five advantages: 1) PGMs for websites can be trained independently from each other. Thus, models for new websites can easily be added, and existing ones can be updated without changing other models, and ProFi does not need samples for the background class, as, e.g., CUMUL does. 2) ProFi can vary the pre-processing steps for each website independently. Varying the pre-processing steps changes the observation spaces for PGMs. Thus, the log-likelihood of PGMs cannot be directly compared since the log-likelihood is not calibrated [80]. The anomaly score solves this problem by quantifying how normal an observation behaves compared to others of the same class. 3) The likelihood of the PGMs can be used to detect changes in the data distribution and thus indicate a reduction in reliability [80]. 4) The PGMs ProFi uses are computationally efficient and can consume data in a streaming manner. That is, the symbols extracted from packets by the Symbolizer can be fed one by one into the models without storing the full sequence.

3.3.5 Defense

A perfect defense would fulfill the following two conditions for a large set of observations $\mathcal{X}$:

$$\eta_y(O) \leq 1 \quad \forall y \in \mathcal{Y} \wedge \forall O \in \mathcal{X}, \tag{3.10}$$

$$\frac{\mid \mathcal{X}_y \mid}{\mid \mathcal{Y} \mid} - \epsilon \leq \sum_{O \in \mathcal{X}_y} \mathbb{I}\left(\min_{y' \in \mathcal{Y}} \eta_{y'}(O) = y\right) \leq \frac{\mid \mathcal{X}_y \mid}{\mid \mathcal{Y} \mid} + \epsilon \quad \forall y \in \mathcal{Y}. \tag{3.11}$$

Specifically, Eq. (3.10) expresses that the defense would result in a value smaller one for the anomaly scores of all websites and sequences. This indicates that the defense removes patterns from the traffic, and sequences of websites look the same for the PGMs. But, although the anomaly scores for all websites are smaller than one, the classifier could still produce the correct label if the website's PGM has the smallest anomaly score. Thus, Eq. (3.11) expresses that the number of times the anomaly score η_y for a website y is the smallest anomaly score for sequences of that website lies within an ϵ-ball of the expected value of a uniform distribution. The classifier's performance is thus similar to randomly sampling a class label.

To achieve this, a defense based on the padding in the TLS record protocol is implemented and evaluated, referred to as Random TLS Record Size Defense (RTLSRS). RTLSRS randomly draws TLS record sizes from a uniform distribution $U(l_{min}, l_{max})$, and pads records to the desired length if necessary. After each record, RTLSRS randomly chooses between adding the next record to the data or handing the current data to the transport protocol. Thus, the sizes and sequence of packets and TLS records change - the features that PROFI uses to detect patterns. The TLS 1.3 [128] standard allows the padding of TLS records to arbitrary sizes up to 2^{14} Bytes. Thus, the defense can readily be implemented in Secure Socket Layer (SSL) libraries.

Characteristics that are not protected with this defense are the TLS handshake and characteristic exchanges between client and server. The handshake protocol does not support padding [128]. Thus RTLSRS leaves the messages as they are. Characteristic exchanges, i.e., a specific sequence of contacts between client and server, could, in principle, be obfuscated with deterministic sending of decoy packets, albeit at the expense of more bandwidth overhead. The evaluations show that the simple defense suffices to thwart PROFI.

The goal of RTLSRS is to be an easy-to-implement and effective defense against PROFI. RTLSRS is not intended to outperform previous WFP defenses. In principle, existing WFP defenses should also effectively mitigate PROFI. However, existing defenses, e.g., HTTPOS [145] or CS-BuFLO [99], are often complex to implement and require continuous effort to maintain, which might hinder their adoption.

3.4 PGMs and baseline attacks

This section explains the used PGMs and how ProFi estimates model parameters from data. In addition, this section introduces three baselines: the k-Nearest Neighbor (kNN) classifier [79] with the Levenshtein distance [162], the SVM-based CUMUL [102] attack, and the set-based IPFP [112] attack. All techniques are trained on a dataset $\mathcal{X}$ consisting of calls to multiple webpages of websites. $\mathcal{X}_{yj}$ denotes the jth webpage of website y.

3.4.1 Markov Chain

ProFi uses a first order homogeneous MC as probabilistic model. A first-order homogeneous MC models a sequence O of length T as:

$$p(O) = p(O_1) \prod_{t=1}^{T-1} p(O_{t+1} \mid O_t). \tag{3.12}$$

The probability of sequence O corresponds to the probability of the sequence starting with the symbol O_1, multiplied by the probability of the next symbol O_{t+1}, given the current symbol O_t. The transition probabilities between symbols are time-invariant [80].

The MC is parametrized with the probabilities of starting with each symbol, and the transition probabilities between each pair of symbols. The parameters are estimated from the training data using maximum likelihood estimation. That is, the estimator counts the occurrences of transitions and normalizes them.

Formally, the probability of a MC for website y to start with a symbol $w \in \mathcal{W}$ is estimated as [27]:

$$p(O_1 = w) \doteq \frac{1}{\mid \mathcal{X}_y \mid} \sum_{O^y \in \mathcal{X}_y} \mathbb{I}(O_1^y = w) \tag{3.13}$$

Similarly, the transition probabilities from a symbol w to v are estimated as [27]:

$$p(O_{t+1} = w \mid O_t = v) \doteq \frac{\sum_{O^y \in \mathcal{X}_y} \sum_{i=1}^{|O^y|-1} \mathbb{I}\left(O_{i+1}^y = w, O_i^y = v\right)}{\sum_{O^y \in \mathcal{X}_y} (\mid O^y \mid -1)} \tag{3.14}$$

To handle unobserved start symbols and transitions, the estimator uses pseudo counts to ensure that each symbol and transition has a small probability of occurring [80].

Fig. 3.9 visualizes a MC modeling MainFlows for the website `www.grammaly.com`. Each node in Fig. 3.9 corresponds to a symbol following the format Sec. 3.3.3 introduces. Arrows between nodes visualize the transition probabilities, while the arrow's thickness visualizes the associated probability. The thicker the line, the more probable the transition. Fig. 3.9 uses a special `start` node to illustrate

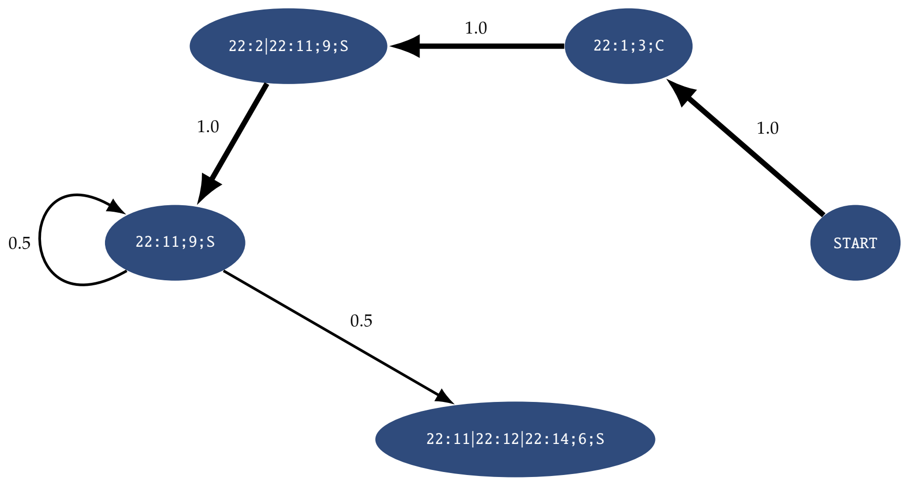


Figure 3.9 MC for `www.grammarly.com`. The edge weights indicate probability. High edge weight corresponds to a high probability. Labels above the arcs give probability. Node labels encode the TLS records and frame sizes.

the start probability in each state. The model uses the MainFlow's first five packets to estimate the parameters and bins the packet's size to 10 bins. The MC in Fig. 3.9 mostly consists of the TLS handshake. The MC starts with a `client_hello` followed by a `server_hello`. Then, the server starts to send a certificate which takes four packets. The first part of the certificate is sent together with the `server_hello` in the second packet. The next two packets contain only the certificate. Note the self-loop on the node `23:11;9;S`. The fifth packet contains the last part of the certificate record, together with `server_key_exchanged` and a `server_hello_done` record.

The equations Eq. (3.13) and Eq. (3.14) show that the MC is parameterized with $| \mathcal{W} |^2 + | \mathcal{W} |$ parameters. Specifically, each symbol $w \in \mathcal{W}$ and each potential transition between any pair of symbols is associated with a probability. However, in practice, a small fraction of all possible transitions occur, as Fig. 3.9 shows. Storing only the transitions that were actually observed can thus reduce the memory requirements. In the case of ProFi, the largest model has 11 882 parameters compared to millions of parameters the full parameterization would have.

The computational complexity in inference is linear in the sequence length. The required computation amounts to the retrieval of the transition probabilities. Thus, it is possible to implement the MC as a streaming algorithm that processes packet by packet. In this case, the cache of the Coordinator in Fig. 3.7 can be integrated into the classifier, which must then store the intermediate values for each flow and the previous symbol.

3.4.2 Profile Hidden Markov Model

A PHMM [160] is a left-right HMM with a specific structure in the hidden states $\mathcal{Z}$. Fig. 3.10 shows this structure. A PHMM has three types of hidden states:

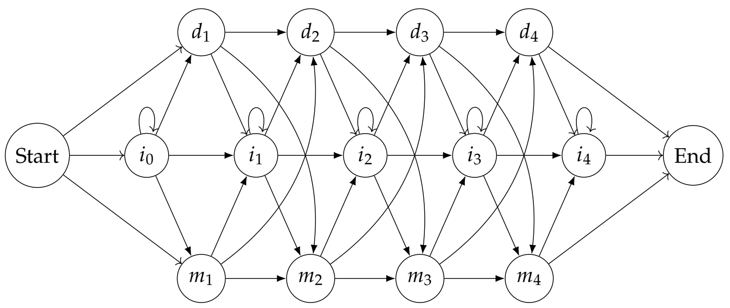


Figure 3.10 Transition between the hidden states of a PHMM. `delete` (D) states do not emit symbols.

`delete` (d), `insert` (i), and `match` (m) states. The `insert` and `match` states emit symbols. The `delete` states are silent, i.e., do not emit a symbol. The PHMM originates in the life sciences and is designed to model genome sequences [160].

The assumption behind the PHMM is that the modeled sequences consist of static and variable parts. The static parts are modeled with the `match` states. The variable parts are modeled with the `insert` and `delete` states. The `insert` states allow the model to insert an arbitrary number of symbols at a specific position. The `delete` states allow the deletion of symbols. In combination, the `delete` and `insert` states allow the replacement of a symbol in the `match` chain. Thus, the PHMM implements a form of probabilistic Levenshtein distance [160].

The PHMM is parameterized by a transition model τ and an emission model o. The transition model gives the probability of moving from one hidden state to another. The initial probabilities are trivial for the PHMM, since all sequences start in the $Start$ state, i.e., for the PHMM, π is trivially defined as:

$$\pi : \mathcal{Z} \to \{0, 1\}, z \mapsto \begin{cases} 1 & \text{if } z = Start, \\ 0 & \text{else.} \end{cases} \tag{3.15}$$

Fig. 3.10 shows that the transition model is sparse for the PHMM since most transitions are not allowed by the structure in the hidden states. For example, Fig. 3.10 shows that it is impossible to move from the `match` state m_1 directly to the `insert` state i_5. The emission model gives for each `insert` and `match` state a probability distribution over possible observations. In ProFi, those are the symbols generated by the Symbolizer.

The model's parameters can be estimated with the Baum-Welch algorithm, an instance of the Expectation-Maximization algorithm [130, 160]. However, the silent `delete` states require special treatment for calculating the forward and backward variables in the Baum-Welch algorithm [160]. The parameter update itself remains unchanged. Since a library implementing the adapted Baum-Welch algorithm was unavailable, this chapter develops an adapted version of the calculation of the forward and backward variables.

The adapted algorithm introduces the forward variables $\alpha_0(d_l)$ for all delete states recursively as:

$$\alpha_0 : \mathcal{D} \longrightarrow [0,1]; d_l \mapsto \begin{cases} \tau(d_1 \mid Start) & \text{if } d_l = d_1 \\ \tau(d_l \mid d_{l-1})\,\alpha_0(d_{l-1}) & \text{else.} \end{cases} \tag{3.16}$$

The forward variables α_0 correspond to the contingency, that the observed sequence is generated through a path through the PHMM that skips an arbitrary number of `insert` and `match` states before the first symbol is generated. The initial forward variables for m_1 and i_0 are defined according to Eq. (3.4) as:

$$\alpha_1(m_1) \doteq \tau(m_1 \mid Start)\, o(O_1 \mid m_1) \tag{3.17}$$

$$\alpha_1(i_0) \doteq \tau(i_0 \mid Start)\, o(O_1 \mid i_0) \tag{3.18}$$

Specifically, the only viable path through a PHMM resulting in either m_1 or i_0 and generating the symbol O_1 is by transitioning from the state $Start$ directly to either of the two hidden states. In contrast to a normal left-right HMM, all other states corresponding to later time-steps are reachable by traversing the `delete` states. The forward variables for the `match` states are defined as:

$$\alpha_1(m_l) \doteq \alpha_0(d_{l-1})\,\tau(m_l \mid d_{l-1})\, o(O_1 \mid m_l) \qquad \forall 1 < l \leq L, \tag{3.19}$$

where $L \in \mathbb{N}$ is the length of the PHMM, i.e., the number of `match` states. The forward variables for the `insert` states are defined as:

$$\alpha_1(i_l) \doteq \alpha_0(d_l)\,\tau(i_l \mid d_l)\, o(O_1 \mid i_l) \qquad \forall 1 \leq l \leq L \tag{3.20}$$

The forward variables for the `delete` states given that the first symbol has been observed are defined as:

$$\alpha_1(d_l) \doteq \begin{cases} \alpha_1(i_1)\,\tau(d_1 \mid i_1) & \text{if } l = 1 \\ \sum_{z \in \{i_l, m_{l-1}\}} \alpha_1(z)\,\tau(d_l \mid z) & \text{else.} \end{cases} \tag{3.21}$$

For the `match` and `insert` state, the forward variables encode the probability of reaching a preceding `delete` state corresponding to the newly introduced $\alpha_0(d_l)$ variables, the probability to transition from the `delete` state to the `match` or `insert` state, and the probability of generating the first symbol of the sequence there. Similarly, since `delete` states do not generate a symbol, the only paths that generate O_1 and end in a delete state start in a `match` or `insert` state where O_1 has been emitted, i.e., correspond to the respective α_1 variables of `match` and `insert` states.

The forward variables for the remaining sequence elements $O_{2:T}$ are similarly defined. The forward variables for the `match` states are defined as:

$$\alpha_t(m_i) \doteq \begin{cases} \alpha_{t-1}(i_0)\,\tau(m_1 \mid i_0)\,o(O_t \mid m_1) & \text{if } i = 1 \\ \sum_{z\in\{i_{i-1},m_{i-1},d_{i-1}\}} \alpha_{t-1}(z)\,\tau(m_i \mid z)\,o(O_t \mid m_i) & \text{else.} \end{cases} \tag{3.22}$$

The forward variables for the `insert` states are defined as:

$$\alpha_t(i_i) \doteq \begin{cases} \alpha_{t-1}(i_0)\,\tau(i_0 \mid i_0)\,o(O_t \mid i_0) & \text{if } i = 0 \\ \sum_{z\in\{i_i,m_i,d_i\}} \alpha_{t-1}(z)\,\tau(i_i \mid z)\,o(O_t \mid i_i) & \text{else.} \end{cases} \tag{3.23}$$

And the forward variables for the `delete` states are defined as:

$$\alpha_t(d_i) \doteq \begin{cases} \alpha_t(i_1)\,\tau(d_1 \mid i_1) & \text{if } i = 1 \\ \sum_{z\in\{i_i,m_{i-1}\}} \alpha_t(z)\,\tau(d_i \mid z) & \text{if } 1 < i \leq T. \end{cases} \tag{3.24}$$

The definition of the backward variables also changes slightly to accommodate the silent `delete` states and the PHMM's left-right nature. The backwards variable for the End state is set statically to one [130]:

$$\beta_T(End) \doteq 1 \tag{3.25}$$

For the remaining hidden states $z_l \in \mathcal{Z}/\{Start, End\}$, the initial backward variables are defined as:

$$\beta_T(z_l) \doteq \begin{cases} \tau(End \mid z_l)\,\beta_T(End) & \text{if } l = L \\ \tau(d_{l+1} \mid z_l)\,\beta_T(d_{l+1}) & \text{else.} \end{cases} \tag{3.26}$$

The formulas can be interpreted as follows: The End state generates a special symbol indicating the end of the sequence. Only the End state can generate this symbol. Thus, after the last element O_T in a sequence has been observed, the end symbol remains. The only way to create the end symbol after all other elements in the sequence have been generated is for i_L and m_L to transition directly to the End state, and for all other states to traverse to the End state via the `delete` states.

For the generation of the sequence's symbols $O_{1:T}$, the backwards variables for the hidden states $z_l \in \mathcal{Z}/\{Start, End\}$ are defined as:

$$\beta_t(z_l) \doteq \begin{cases} \tau(i_L \mid d_L)\,o(O_t \mid i_L)\,\beta_{t+1}(i_L) & \text{if } l = L \\ \sum_{z'\in\{d_{l+1},i_l,m_{l+1}\}} \tau(z' \mid z_l)\,o(O_l \mid z')\,\beta_{t+1}(z') & \text{else.} \end{cases} \tag{3.27}$$

Here, the states i_L, m_L, and d_L need special treatment. As long as there are still symbols left, the PHMM cannot transition to the end state. Thus, the only predecessor states of i_L, m_L, and d_L is i_L.

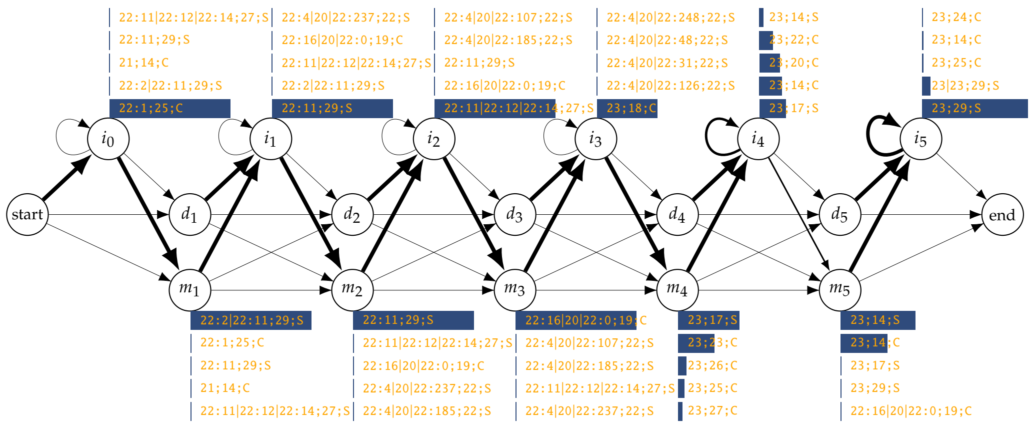


Figure 3.11 Trained PHMM for `www.grammarly.com`. The figure indicates self-loops on the `insert` through a separate node prefixed with s. The figure shows the five emissions for `match` and `insert` states with the highest probability. The figure indicates the probability through a bar. The figure indicates the transition probability between hidden states through the edge weight, where a higher weight indicates a higher probability.

Once the parameters are estimated, the log-likelihood of the PHMMs can be computed with the Forward-Algorithm [130], which is also used in the Baum-Welch Algorithm [130]. The computational complexity of the forward algorithm for the PHMMs grows cubically with the length of the PHMMs. The memory complexity is smaller compared to the MC. In total, a PHMMs of length L has $(2L + 1) \mid \mathcal{W} \mid +9L + 3$ parameters.

Fig. 3.11 illustrates the estimated parameters of a PHMM for the website `www.grammarly.com`. The PHMM is fitted on sequences generated by the symbolized from the first 10 packets of MainFlows from `www.grammarly.com`. The PHMM in Fig. 3.11 has a length of $L = 5$. Fig. 3.11 illustrates the transition probabilities through the line thickness. The thicker the line, the higher the corresponding transition probability. Fig. 3.11 illustrates the emission model by listing for each `insert` and `match` state five symbols with the highest emission probabilities. The symbol names follow the format Sec. 3.3.3 introduces. The Symbolizer bins each packet's length to 30 bins using Logarithmic binning. Like Fig. 3.9, the first five packets correspond to the TLS handshake. The arcs in Fig. 3.11 show a straight path through the hidden states, alternating between `match` and `insert` states. Alternating between `match` and `insert` states allows the PHMM to produce more symbols compared to, e.g., traversing along the `match` states. After the PHMM emitted the `server_hello_done` message in state i_2, the PHMM produces a packet with `client_key_exchange`, `change_cipher_spec`, and `hello_request` records send by the client in state m_3. The PHMM then transitions to i_3, generating a packet containing application data sent by the client. From i_3, the PHMM transitions to m_4. The emission model of m_4 is broader. The symbol with the highest probability corresponds to a packet sent by the server containing application data. The other four symbols correspond to application data with varying sizes sent by the client. Thus, the sixth packet in the Main-

Flows of `www.grammarly.com` have roughly equal chances to come from the server or the client. Then, the PHMM transitions to state i_4, also having a broader emission model. Again, the emissions correspond to application data sent by the client or the server. In contrast to previous `insert` states, i_4 assigns a probability of 0.61 to its self-loop. The PHMM in Fig. 3.11 uses i_4 to generate multiple symbols. With a probability of 0.38, the PHMM in Fig. 3.11 transitions from i_5 to m_5, where the PHMM emits a symbol corresponding either to a packet carrying application data from the server to the client, or the client to the server. Then, the PHMM transitions to i_5 where the PHMM remains with a probability of 0.87 and transitions with a probability of 0.13 to state End. The state i_6 generates packets with a length falling into bin 29 containing one or two records with application data traveling from the server to the client.

3.4.3 k-Nearest Neighbor

kNN is an instance-based learning method and is used in several previous WFP attacks [97, 98, 101]. This chapter thus uses the kNN classifier as a baseline for ProFi.

In instance-based learning, the training data is stored verbatim [79]. During inference, i.e., for predicting a class label, kNN uses a distance function to retrieve the k nearest neighbors of a test sample. kNN returns the label that occurs most often among the k neighbors[79].

As in [98], kNN uses the Levenshtein Distance [162] as distance measure. The Levenshtein Distance is a string metric that measures the difference between sequences: $d_L : 2^{\mathcal{W}} \times 2^{\mathcal{W}} \rightarrow \mathbb{N}$. Here, the strings correspond to sequences of symbols extracted by the Symbolizer from the first n packets of a flow. A symbol, i.e., representation of a packet or record, corresponds to a single character in a string. To classify a new flow, kNN computes the Levenshtein Distance between the extracted packets and all sequences in the training set.

In the open-world scenario, this approach is not applicable since kNN always returns a label. However, in the open-world scenario, kNN must reject samples. To achieve this, kNN labels a flow as background if the number of neighbors with the most frequent label is below a certain threshold, similar to previous work [98]. For example, let $k = 9$ and the neighbors for a test instance belong to seven websites. Further, let the number of neighbors that must agree be 5. Now, two neighbors belong to one website, and the other six neighbors belong to six other websites. In this scenario, the majority label has two out of nine votes, less than the required five agreeing neighbors. kNN thus rejects the majority label and instead classifies the sample as background traffic. This approach has the advantage that no additional data is required to represent the background class.

For inference, kNN induces memory and computational costs that increase linearly with the training set size. Here, the computational complexity of classifying one sample is $\mid \mathcal{X} \mid \cdot (l \cdot m + 1)$, where $\mid \mathcal{X} \mid$ is the number of training samples, l is the length of the first sequence, and m is the length of the second sequence. The computational complexity of the Levenshtein Distance is $l \cdot m$. The distance must be calculated for every training sample. After that, the k nearest neighbors must be retrieved, which requires a scan through all calculated distances. Due to the nature of the underlying data, methods such as *kD-trees* [79], or *ball trees* [79] are not applicable.

3.4.4 CUMUL

The CUMUL attack [102] uses SVMs as classifiers and features that CUMUL derives from all packets that belong to a webpage call. CUMUL forms a sequence of packet sizes $T = (p_1, \dots, p_N)$, where $p_i > 0$ indicates inbound packets and $p_i < 0$ outbound packets. Then, CUMUL forms two cumulative representations, $A(T) = (0, a_1, \dots, a_N)$ and $C(T) = (0, c_1, \dots, c_N)$ from T, where $a_1 =\mid p_1 \mid$, and $c_1 = p_1$. Then, $a_i = a_{i-1} + \mid p_i \mid$, and $c_i = c_{i-1} + p_i$. CUMUL derives the input to the SVM from $A(T)$ and $C(T)$ by concatenating the piece-wise linear interpolations of n equidistant points, resulting in a fixed length representation of a webpage call [102].

In contrast to PHMMs, MC, and kNN, CUMUL is a pure multi-class classifier. To operate in an open-world scenario, the training set must contain samples for the background traffic, i.e., CUMUL explicitly needs data from websites not in the set of websites that should be surveyed. The computational complexity of the SVM in inference time is linear in the number of support vectors identified during training [79].

3.4.5 IPFP

IPFP [112] uses the IP addresses that occur while loading a website's webpages to construct a fingerprint. Thus, an adversary first browses a targeted website and notes all domain names contacted during that time. The attacker resolves the domain names to IP addresses. During the attack, the adversary compares the IP addresses that occur during a victim's page load and compares those to the previously recorded IP addresses. The attacker then labels the page load as the website with the highest certainty given the observed IP addresses.

To perform the attack, the authors differentiate between primary domains and IP addresses and secondary domains and IP addresses. Primary domains correspond to the domain in the URL showing in the browser's address bar while browsing the website. Primary IPs correspond to all IPs a primary domain resolves to. Secondary domains correspond to all other domains that occur while browsing the website, and the secondary IPs to the IP addresses the secondary domains resolve to. In this way, the attacker obtains for each website $y \in \mathcal{Y}$ a set of primary domains $\mathcal{F}_D^y$, IPs $\mathcal{F}_{IP}^y$ and a set of secondary domains $\mathcal{S}_D^y$, and IPs $\mathcal{S}_{IP}^y$ [112].

In an intermediate step, the attacker then computes for each observed domain $d \in \bigcup_{y \in \mathcal{Y}} \mathcal{S}_D^y$ the entropy

$$\text{entro}_D(d) \doteq -\log_2\left(p(d)\right) \tag{3.28}$$

of the probability $p(d)$ that this domain will be contacted when visiting an arbitrary website. Specifically, the probability of observing domain d for an arbitrary website is defined as [112]:

$$p(d) \doteq \frac{\sum_{y \in \mathcal{Y}} \mathbb{I}\left(d \in \mathcal{S}_D^y\right)}{\mid \mathcal{Y} \mid}. \tag{3.29}$$

Then, the attacker brings the domain's entropy to the IP level by taking the average of the entropy of domains the IP resolves to. This accounts for the fact that a single IP can host multiple domains. Assuming the helper function:

$$\text{domain} : \bigcup_{y \in \mathcal{Y}} \mathcal{S}_{IP}^y \rightarrow 2^{\bigcup_{y \in \mathcal{Y}} \mathcal{S}_D^y} \tag{3.30}$$

mapping from an IP $a \in \bigcup_{y \in \mathcal{Y}} \mathcal{S}_{IP}^y$ to the domains that resolve to a. Then, the entropy of an IP a is defined as [112]:

$$\text{entro}_{IP}(a) \doteq \frac{\sum_{d \in \text{domain}(a)} \text{entro}(d)}{\mid \text{domain}(a) \mid} \tag{3.31}$$

To classify website accesses, the attacker requires the sequence of IP addresses $O = [ip_1, ip_2, \ldots, ip_N]$ the client contacts when retrieving a webpage. The attacker then constructs a set of candidate websites $\mathcal{Y}'$ by matching ip_1 to each website's set of primary IP addresses [112]:

$$\mathcal{Y}' \doteq \bigcup_{y \in \mathcal{Y} : ip_1 \in \mathcal{F}_{IP}^y} \{y\}. \tag{3.32}$$

If $\mathcal{Y}'$ is empty, the website is not in the monitored set. If $\mathcal{Y}'$ is nonempty, the attacker retrieves the label $\hat{y}$ resulting in the largest sum of entropy values for the IPs in the sequence $O_{2:N}$ [112]:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}'} \sum_{i=2}^{N} \mathbb{I}\left(ip_i \in \mathcal{S}_{IP}^y\right) \text{entro}_{IP}(ip_i). \tag{3.33}$$

Intuitively, the website resulting in the largest sum of average entropies for the monitored IP address can be interpreted as being the website with the highest certainty of having generated the observed IP addresses.

IPFP is very sensitive to the extraction of the full page load. For example, if the first IP address is not retrieved, the classification procedure most likely fails. Further, the classification approach assumes a form of closed-world scenario.

IPFP cannot reject a page load as not belonging to any of the monitored websites if the initial IP is a primary IP of one of the monitored websites. Similarly to a MLC, IPFP will produce a label in this situation.

3.5 Data Acquisition and Model Training

This section describes the method used to obtain training data and how models are trained. Thus, this section reflects MaLANe's *Analyze Networked System*, *Generate Data*, *Prepare Training*, *Investigate Data*, and *Train Model* activities. Sec. 3.5.1 describes the data acquisition. Sec. 3.5.2 describes how the data is prepared for training and how the models are trained. Sec. 3.5.3 explains each classifier's hyperparameters and how the space of hyperparameters is searched. Sec. 3.5.4 presents the selected hyperparamters, and Sec. 3.5.5 summarizes this section.

3.5.1 Data Acquisition

To evaluate the attacks, traces for 96 websites located in three popular CDNs are gathered. For each CDN, the top 30 websites based on the Alexa Top 1000 [163][7] ranking are selected. For each website, 50 random sub-pages were selected using a javascript-enabled web-crawler. This process results in a data set consisting of 4 800 webpages. To obtain traffic samples, each webpage is accessed on 70 days from the 9th April 2021 until the 24th June 2021. Every website is accessed with the Chromium and the Firefox browser in headless mode. Two browsers were used since websites are known to behave differently for different browsers [164]. This resulted in 100 samples for each website and day, i.e., 9 600 traces per day. More than 3 TB of traces were collected.

To collect the traces, docker containers [165] were executed on three physical machines running Ubuntu 20.04. The docker containers further isolated traffic of webpage accesses and maintained equal conditions for all pages. A browser is started from inside the docker container in headless mode. Each webpage is traced for 7 s, after which the docker container is terminated. Traffic was collected inside the docker container with the `tcpdump` utility. During tracing, NIC-offloading features such as TCP segmentation offloading were disabled.

The gathered data is limited because it does not contain mobile traffic and webpages requiring a previous login. This type of traffic is not included since its acquisition requires tools such as Selenium or virtualized environments that could impact the feature distribution. Websites are known to detect such technologies, potentially blocking or responding in different ways than normal [166]. Further, to log in, users normally have to access a publicly accessible landing page first. Similarly, no traces obtained through passive listening to real traffic were used since no access to such a monitoring location was available.

[7] As of May 1st, the Alexa service is no longer available.

Parameter	Values
Binning Method	EqualWidth, Logarithmic, None, SingleBin
Num Bins	$\{0, 1\} \cup \{10 \cdot i \mid i = 1, \ldots, 10\}$
Num Packets	$\{5 \cdot i \mid i = 1, \ldots, 6\}$
Sequence Element	`packet`, `record`
kNN num Neighbors	$\{3, 6, 9\}$
PHMM length	$\{5 \cdot i \mid i = 1, \ldots 6\}$
CUMUL c	$\{2^{11}, \ldots, 2^{18}\}$
CUMUL γ	$\{2^{-3}, \ldots, 2^{4}\}$

Table 3.3 Hyperparameters for the preprocessing of data and each classifier.

3.5.2 Data Split and Training Procedure

MC, PHMM, kNN, CUMUL, and IPFP are evaluated in a closed-world and an open-world scenario. The websites are divided into two sets: *Foreground Sites* and *Background Sites*. The *Foreground Sites* are the sites that the classifiers should detect. The *Background Sites* are the sites that should be ignored. In the closed-world scenario, the classifiers must differentiate the pages in the *Foreground Sites*. In the open-world scenario, the classifiers must detect whether traffic belongs to a page from the *Foreground Sites*, and if so, from which one. The sets were created by randomly assigning websites to one of the two sets, such that each website type is represented in each set. For example, both sets contain sites with adult content, news pages, etc.

The classifiers are trained on samples from all 70 days from the *Foreground Sites*. For training, webpages of a website are split into three disjoint sets: training, validation, and test set. The training set contains 60 % of the webpages, the validation and test sets contain 20 % each. For hyper-parameter optimization, the models are fit to all traces in the training set, and the performance is evaluated on all 70 days in the validation set. For CUMUL, a background class is included in the open-world scenario as suggested in [102]. The background class consists of traces of one webpage from every website in the *Background Sites*. For the final evaluations, parameters were selected, resulting in the best *precision* on the validation set. That is, models with fewer false positives are preferred. For the final results, the classifiers are trained on all traces from the training and validation set with the selected hyperparameters and evaluated on all 70 days in the test set.

3.5.3 Hyperparameter optimization

Each classifier's hyperparameters are optimized with a grid search. Tbl. 3.3 lists the parameter space. The Binning Method, number of bins, number of packets, and sequence element is optimized for PHMM, MC and kNN. The binning methods map a packet's or TLS record's size to a discrete bin. EqualWidth binning uses equally spaced bins. The bin sizes of Logarithmic binning increase expo-

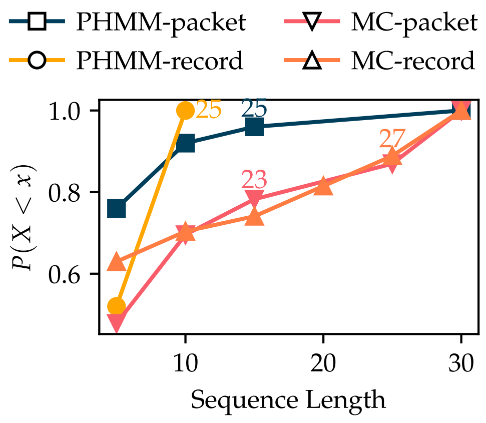


Figure 3.12 Cumulative Distribution Function (CDF) of the discrete sequence lengths and sequence types of PHMM and MC. Numbers in the figure give the total count of models that use the corresponding sequence type.

nentially, i.e., the logarithm of the bin sizes corresponds to equally spaced bins. SingleBin maps all sizes to a single bin, effectively removing the size information from the symbol. None binning does not apply any binning and takes the size verbatim.

For kNN, the number of neighbors is varied, and for the PHMM, its length is varied. For CUMUL, the hyperparameter space of the SVM suggested in [102] is explored. IPFP has no hyperparameters.

In the case of kNN, the same symbolizer σ is applied to MainFlow packets from all websites. That is, the same sequence element from the same number of packets from all MainFlows of all websites are mapped to a symbol with the same binning method and number of bins.

MC and PHMM take advantage of the fact that models for websites are independent, thus varying the symbolizer σ for every website, i.e., each model has its own function σ. A flow U is converted to a unique sequence for every model. With this approach, it is possible to tune the conversion to symbols in such a way that the PGMs can model them well, and at the same time, differentiate them from sequences of other websites. The parameters of the model are estimated *only* on the samples of the corresponding website.

3.5.4 Selected hyperparameters

The best configuration for kNN uses 9 neighbors and the size and direction of the TLS records in the first 30 packets of the MainFlow. The classifier has 40 bins with record sizes of EqualWidth binning. The best configuration for CUMUL is $\gamma = 2^{11}$ and $c = 2^3$.

MCs and PHMMs have diverse configurations, indicating that it is beneficial to tune the pre-processing for each website independently. Of special interest are the selected sequence lengths and sequence elements impacting the computational cost of the attack. Longer sequences require more packets and more processing.

Relying on TLS record information requires the additional parsing of TLS headers. Fig. 3.12 shows the CDFs for the best PHMM and MC models. The CDFs show the distribution over the sequence lengths and sequence elements. The numbers in Fig. 3.12 indicate how many models used the respective sequence type. Fig. 3.12 shows that 25 PHMMs use records and 25 PHMMs use frames. The PHMMs using records require up to 10 packets only, i.e., the records contained in the first 10 packets of each MAINFLOW. PHMMs using frames require 5, 10, 15 and 30 packets. For the MC models, 23 use frames, and 27 records as sequence elements. The length of sequences varies. For both, MC and PHMM, more than 50 % of the models require only five packets. Note that the models with TLS records as sequence elements can get sequences longer than the number of packets, since one packet can carry more than one TLS record, especially during the handshake.

The fact that many MC and PHMM models rely on the first 5 packets is surprising. The first 5 packets contain mostly the TLS handshake. Depending on the TLS configuration, the handshake packet and record sizes vary. For example, servers and clients can have different extensions and exchange certificates of varying sizes [128]. Fig. 3.8 illustrates this, showing that even for a single website, the TLS handshake varies. The differences between the websites are enough for the PGMs to distinguish websites from each other.

3.5.5 Summary

PGMs can represent the MAINFLOWS of websites. The results show that many websites can be distinguished with as little as the first 5 to 10 packets. This is in contrast to previous work on WFP, which relies on all packets sent during the page load. Future work on PGM-based attacks could improve the emission model and address potential multi-modal behavior in the data. Currently, the models rely heavily on the NONE binning method, particularly models for websites with dynamic content. The NONE binning method requires many training samples to get sufficient data since the packet and record sizes are not compressed, i.e., every size that occurs normally must be captured. Here, continuous emission models might better compress the emissions and reduce the risk of overfitting and the number of required samples. The mix of discrete and continuous features makes improving the emission model challenging. Similarly, the data shows that webpages can express multi-modal behavior. This poses a challenge to the used PGMs since their ability to capture multi-modal sequences is limited. Investigating the performance of mixture models, e.g., a mixture of MCs and PHMMs to model one website, could be an interesting aspect for future work.

The attack can likely be thwarted by randomizing or standardizing packet and record sizes due to the reliance of the models on packet and record sizes since SINGLEBIN reduces the average precision to only 20 % on the validation set. Performance decreases because SINGLEBIN removes the packet and record size information, leaving only direction and TLS record type as information. Randomizing packet and record sizes has a similar effect since randomizing essentially removes size information from the data. Furthermore, the large number of models that work well with only five packets indicate that the TLS handshake is important for classification. Standardizing the TLS handshake, i.e., handshakes use the same sequence and size of packets independent of the concrete TLS settings, could help mitigate WFP as well.

3.6 Evaluation

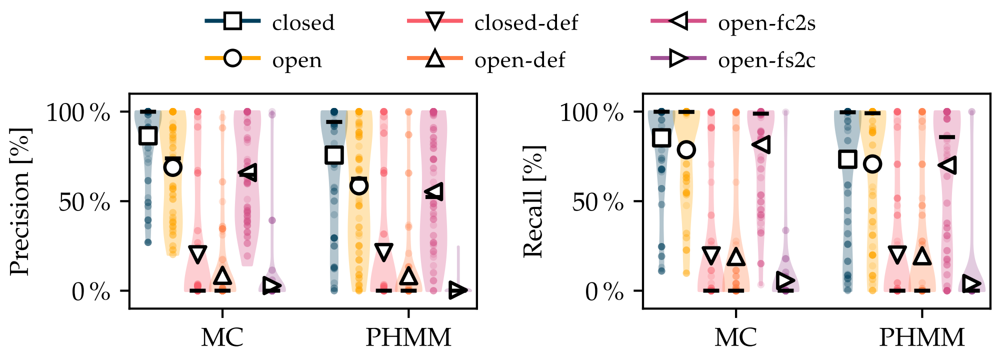
The evaluation focuses on the metrics precision $\doteq \frac{TP}{TP+FP}$ and recall $\doteq \frac{TP}{TP+FN}$, where TP are the true positives, FP the false positives, and FN the false negatives [79]. High precision means reliability, i.e., retrieved instances usually belong to the predicted class. High recall means that the classifier retrieves most of a class's samples. Usually, there is an interdependence between precision and recall, i.e., improving a classifier's recall comes at the cost of reduced precision, and vice versa. ProFi's classifier can trade precision for recall by adjusting the parameter γ appropriately. This section does not discuss the accuracy of the classifiers since the accuracy has little expressiveness. With almost 100 classes, a classifier can achieve 99 % accuracy by predicting one class for all samples [79], analogous to classification problems with imbalanced classes. Here, precision and recall better reflect a classifier's performance.

This section shows results for ProFi's classifier in an open- and closed-world scenario, with and without defense, and under asymmetric routing. Further, this section compares ProFi's classifier against three baselines. This section evaluates RTlsRS by sampling record sizes from a uniform distribution with a lower limit of 100 and an upper limit in the set $\{100i + 100\,\text{Bytes} \mid i = 0, \ldots, 16\}$.

With asymmetric routing, packets traveling from server to client can take a different route than packets from client to server [167]. Thus, ProFi might not observe both directions. Since asymmetric routing frequently occurs in Wide Area Networks (WANs) [167], evaluating ProFi under its influence is important.

This section compares ProFi's performance to that of three previous attack models: kNN, CUMUL [102], and IPFP [112]. kNN has been used in previous attacks [98, 101, 103] and operates on the same feature space as ProFi, i.e., on the sequence of packets and TLS frames. kNN stores the training data verbatim and is a strong baseline [79]. CUMUL [102] is one of the most cited WFP attacks and uses an SVM and traffic features from the full page load. IPFP [112] is a recent attack leveraging the IP addresses that occur during the page-loads to form fingerprints.

kNN, CUMUL, and IPFP are not directly applicable in the attack scenario in Sec. 3.3. kNN has a large memory overhead and is costly to evaluate even with appropriate data structures. CUMUL and IPFP require access to all flows of a page load, making them hard to use with traffic aggregates. Especially IPFP relies heavily on correctly extracting page loads from traffic aggregates, a mostly open



(a) Precision. **(b)** Recall.

Figure 3.13 Precision and recall for the closed-, and open-world scenario, with defense (-def) scenarios, filtered traffic from server to client (-fs2c), and from client to server (-fc2s).

problem. In contrast to kNN, CUMUL's SVM is not suited to model variable-length sequences with nominal elements.

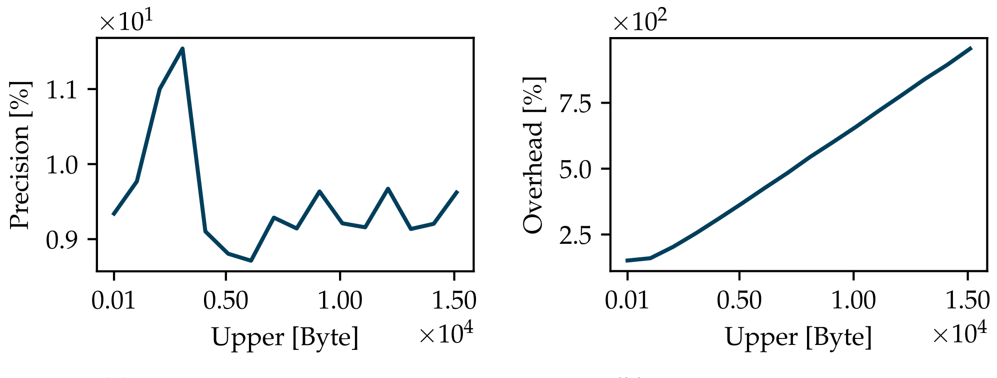
3.6.1 Closed World vs. Open World.

Fig. 3.13 shows violinplots for precision and recall for the open- and closed-world scenarios with and without defense. Asymmetric routing is indicated with the endings `fc2s` and `fs2c`. For `fc2s`, the traffic from the client to the server is unobserved. For `fs2c`, the traffic from the server to the client is unobserved. Markers indicate the average, horizontal bars the median, and small dots are actual samples. Each sample corresponds to the average for one website in the *Foreground Sites* across all days. For example, a sample with 95 % recall means that the model missed 5 % of a webiste's webpages across all days.

MC and PHMM achieve better precision and recall in the closed-world scenario, which is expected. In the closed-world scenario, MC achieves an average precision of 86.5 % and an average recall of 85.4 %, and the PHMM of 75.7 %, and 73.4 %, respectively. In the open-world scenario, the precision and recall reduce to 68.9 % and 78.7 % for MC, and 58.6 % and 70.8 % for PHMM. However, the distribution of precision and recall is skewed, as Fig. 3.13 shows. Both models achieve a median precision and recall of > 99 % in the closed-world scenario. The median precision for MC in the closed-world scenario is 99.9 %, and 94.2 % for the PHMM. In the open-world scenario, the median precision drops for both models close to the average. This indicates that the PGMs confuse *Background Sites* with *Foreground Sites*. Still, both models have a precision of > 90 % for one-third of the *Foreground Sites* in the open-world scenario.

3.6.2 RTLsRS defense.

Fig. 3.14 shows the precision and the overhead as a function of the uniform distribution's upper size. Fig. 3.14a shows that the precision is low for all upper limits, ranging between 8 % and 12 %, with a minimum of 8.71 % for an upper size



(a) Upper limit vs. precision. **(b)** Upper limit vs. overhead.

Figure 3.14 Line plot showing precision and overhead as a function of RTlsRS' upper sizes.

of 6 100 Bytes. In contrast, Fig. 3.14b shows that the overhead increases linearly with the upper size. An upper size of 100 Bytes has the smallest overhead with 150 %. The upper size of 6 100 Bytes has an overhead of 425 %. This section thus shows RTlsRS' effect on the classification performance for an upper size of 100 Bytes. This reduces the average precision to 9.35 % while keeping overhead low.

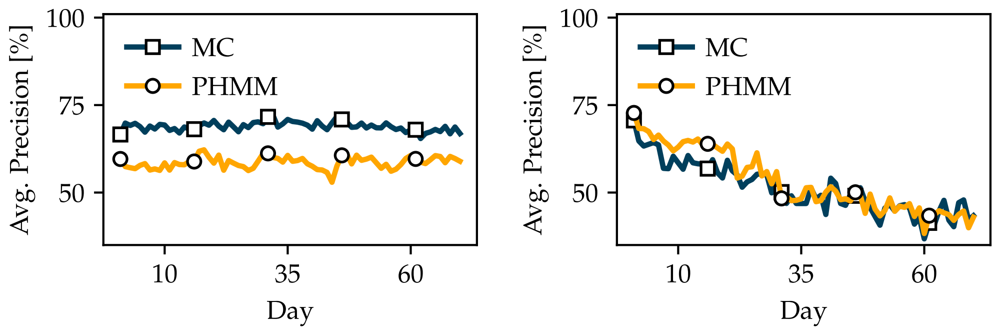
Fig. 3.13 shows that the RTlsRS defense from Sec. 3.3.5 effectively mitigates ProFi. The RTlsRS defense reduces the precision and recall of both models to ~20 % in the closed-world scenario. In the open-world scenario, RTlsRS reduces the precision of both models to less than 10 %. The recall is less affected and reduces to ~19 %.

The RTlsRS defense thus has two effects: The models miss many webpages (low recall), and the credibility of the retrieved webpages is small (low precision). On average, more than 90 % of the webpages the models classify as belonging to a specific website belong to another. Still, even in the presence of RTlsRS, a few websites exist for which ProFi achieves high precision and recall, as Fig. 3.13 shows.

3.6.3 Asymmetric routing.

Fig. 3.13 shows that the effect of asymmetric routing on ProFi's performance depends heavily on the direction of unobserved traffic. If traffic from the server to the client remains unobserved, the average precision and recall of MC and PHMM deteriorate to less than 6 %.

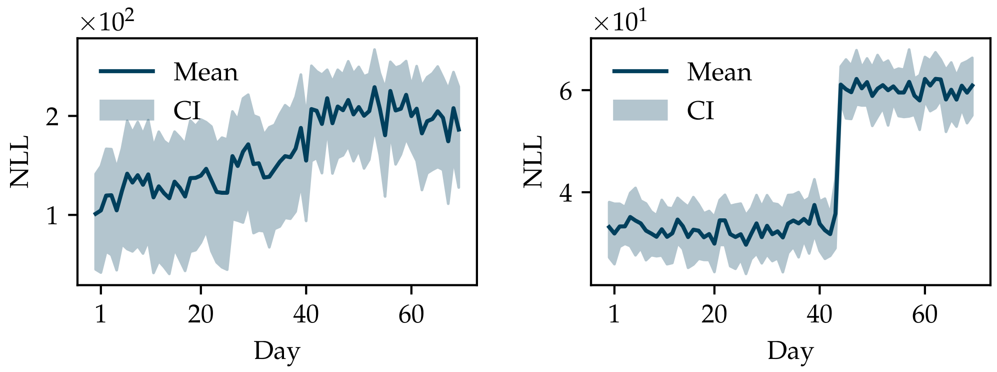
If traffic from the client to the server remains unobserved, then the average precision of MC and PHMM reduces by ~3 %. Similarly, the distribution of the recall for the PHMM shifts towards lower values, as Fig. 3.13 shows. The median recall reduces by >13 %, while the average reduces by 0.7 %. For the MC, the average recall improves from 78.7 % to 81.6 %, while the median stays high at 98.9 %.



(a) Precision over days.

(b) One day of training data.

Figure 3.15 Development of precision over multiple days.



(a) `www.medium.com`.

(b) `www.ebay.co.uk`.

Figure 3.16 Development of the Negative Log Likelihood (NLL) for the MC trained on one day. The shaded area is the bootstrapped 99 % CI of the mean.

This outcome is reasonable. Few packets travel from client to server, amounting to TLS messages and Hypter Text Transfer Protocol (HTTP) commands with small variance, making websites hard to differentiate. Most packets travel from server to client and have a high variance, as Fig. 3.8 shows. Missing traffic from the server to the client deprives the PGMs of much of the information they use for classification.

3.6.4 Training Data Evaluation

Fig. 3.15a shows the average precision over days when training on all training data. Fig. 3.15a shows that the precision of all models remains approximately constant over time. The recall scores show similar behavior.

Fig. 3.15b shows the average precision per day when training on traces of the first day. Fig. 3.15b shows that the precision decreases over time. The average precision of MC and PHMM drops to 66.2 % and 42.5 %. This behavior is not unexpected and has previously been observed [132]. This indicates that attackers must continuously gather new data to update the deployed models for websites with dynamic content.

		PHMM	MC	kNN	CUMUL	IPFP
Closed	Precision	75.74	86.51	91.80	63.43	83.68
	Recall	73.40	85.35	91.44	62.18	85.17
Open	Precision	58.58	68.90	61.89	30.74	80.60
	Recall	70.76	78.71	88.71	60.97	85.19

Table 3.4 Precision, recall and accuracy for PHMM, MC, and the baselines in the open- and closed scenario.

Fig. 3.16 illustrates this and shows the average NLL[8] and its 99 % Confidence Interval (CI) for the MCs fitted to data from `www.medium.com` (Fig. 3.16a), and `www.ebay.co.uk` (Fig. 3.16b). Fig. 3.16 shows two examples of distributional drift that can occur in an operative system: A slow and steady drift and a sudden change.

Fig. 3.16a shows that the average NLL has an increasing trend indicating that the input diverges from the data the model was fitted with. To continuously operate the system, the adversary thus has to continuously update its training data and model to track the drift in the input's data distribution.

In contrast, Fig. 3.16b shows a sudden jump after 42 days. This indicates a persistent change in the input data's distribution, making the model unreliable. The change is caused by a different length of the server's TLS `finished` record. If such a change occurs, the adversary must collect new samples to adjust the model to the new input distribution.

The advantage of PGMs is that they readily detect drift in the input's distribution. Further, PGMs allow investigating what changed [80, 156]. Here, the PGMs allow the identification of the transitions with low probability, directly pinpointing the change in the exchanged messages.

3.6.5 Comparison to baselines.

Tbl. 3.4 shows the average precision and recall for MC and PHMM, and the three baselines kNN, CUMUL, and IPFP in the closed-, and open-world scenario. Tbl. 3.4 shows that ProFi is competitive to state-of-the-art models. In the closed-world scenario, kNN is the best model with an average precision of 91.8 % and an average recall of 91.44 %. kNN achieves this at the expense of increased memory and computational requirements, whereas ProFi relies on computationally cheap ML models with few parameters. Fig. 3.1 illustrates this fact and shows the number of pages per second classifiers classify in an offline setting. Fig. 3.1 shows

[8]The NLL is uncalibrated and cannot be compared between models.

that kNN achieves short of 10 classifications per second, whereas MC and PHMM achieve between 30 000 and 100 000 classifications per second[9].

In the open-world scenario, IPFP is the best model with a precision of 80.6 % and a recall of 85.2 %. In contrast to ProFi, IPFP requires additional pre-processing steps to extract a user's page load from the traffic aggregate. The authors do not provide a solution to this problem. Detecting for one user whether the user loads more than one page and detecting the onset of the second page load is already a challenging task [101, 104, 111, 118]. Extracting one page load from traffic aggregates with high precision is much more complex and will have limited accuracy in practice.

3.6.6 Summary

The results show that PGMs classify traffic based on the first 5 to 30 packets of the first initiated TLS session when retrieving a webpage and are competitive to state-of-the-art attacks. This simplifies WFP attacks since page loads do not have to be retrieved from a mix of traffic. Furthermore, the evaluation shows that the attack works with only the traffic from the server to the client, i.e., ProFi can handle asymmetric routing. In addition, the computational and memory complexity of the PGMs is low and could allow adversaries to perform WFP in real-time and at scale. The results show that the PGMs have a higher recall than precision. In scenarios where the precision is of interest, the attacker can trade recall for precision by setting the classifier's parameter γ to a value smaller one. A value smaller one requires a log-likelihood at inference time that is smaller than the largest log-likelihood at training time, i.e., for which the learned model fits better than the worst model during training. Fortunately, the RTlsRS defense that SSL libraries can readily implement thwarts the attack at the cost of a 150 % increase in bandwidth.

Concerning performance improvements, an adversary could easily improve ProFi by conditioning the probability of a website on additional variables such as the visited CDN, the time of the day, or other correlated variables. For example, calculating the probability for a website y could then be $p(y \mid O)p(y \mid c)p(y \mid t)$, where c identifies the CDN the website is located on and t the time of the day. For example, conditioning on the CDN would essentially reduce the hypothesis space to just the websites in the CDN. Extending the classifier with this information could be an interesting avenue for future work.

[9]Data structures to improve the inference time, e.g., KDTrees, are not applicable since they require real-valued inputs [168]. The input consists of sequences with nominal elements. Further, experiments using the packet size of the first 30 packets as input features showed that the KDTree reduced the inference time not as strongly as expected, i.e., only by a factor of three compared to a linear scan. This is not unexpected [168].

3.7 Prototype Design & Results

This section presents a microservice-based prototype that implements the ProFi attack. This section corresponds to MaLANe's *Integrate Model* and *Deploy Model* activities. The prototype is based on readily available open-source software and Commodity of the Shelf (COTS) hardware to showcase that ProFi is applicable in practice and give an impression of the scale WFP attacks might operate at, e.g., by operating on a MEC infrastructure. After establishing the ML-related performance in the previous section, this section's evaluation focuses on networking-related performance indicators.

3.7.1 Design & Architecture

This section first justifies the design decisions and then explains the architecture in detail.

Design

Network Function Virtualization (NFV) and COTS hardware. The principle of NFV is a good fit for ProFi. NFV can realize each PGM as one VNF. NFV enables the scaling of ProFi across multiple servers, horizontal scaling of expensive PGMs, and the deployment of new, or updating of existing PGMs at runtime. This makes ProFi amenable to the deployment on top of MEC infrastructure in combination with containerization and orchestration tools such as K8S. The low computational cost of the PGMs alleviates the need for special hardware accelerators and lowers the burden of deploying ProFi in practice.

Middlebox character. Despite being microservice-based, ProFi has a middlebox character. This enables the attacker to interfere with a victim's traffic. The prototype could drop the remaining packets of a flagged flow. The alternative, i.e., mirroring traffic to an analysis server, does not easily allow this type of interference since switches start to sample traffic at high rates, which will result in incorrect classifications.

Architecture

The prototype is implemented on top of the `OpenNetVM` platform [169]. `OpenNetVM` is a high-performance, zero-copy NF platform supporting containerization that simplifies the development of VNFs . The prototype implements the classification procedure from Fig. 3.7 using a microservice-based architecture. Fig. 3.17 shows the structure of the prototype. The prototype maps the procedure in Fig. 3.7 into five services: The `TLSFilter`, `TLSRecDet`, `Symbolizer`, `PGM`, and `Coordinator`.

VNFs communicate with each other using UDP packets. This makes the deployment of ProFi to a cluster simple. The microservice-based architecture enables

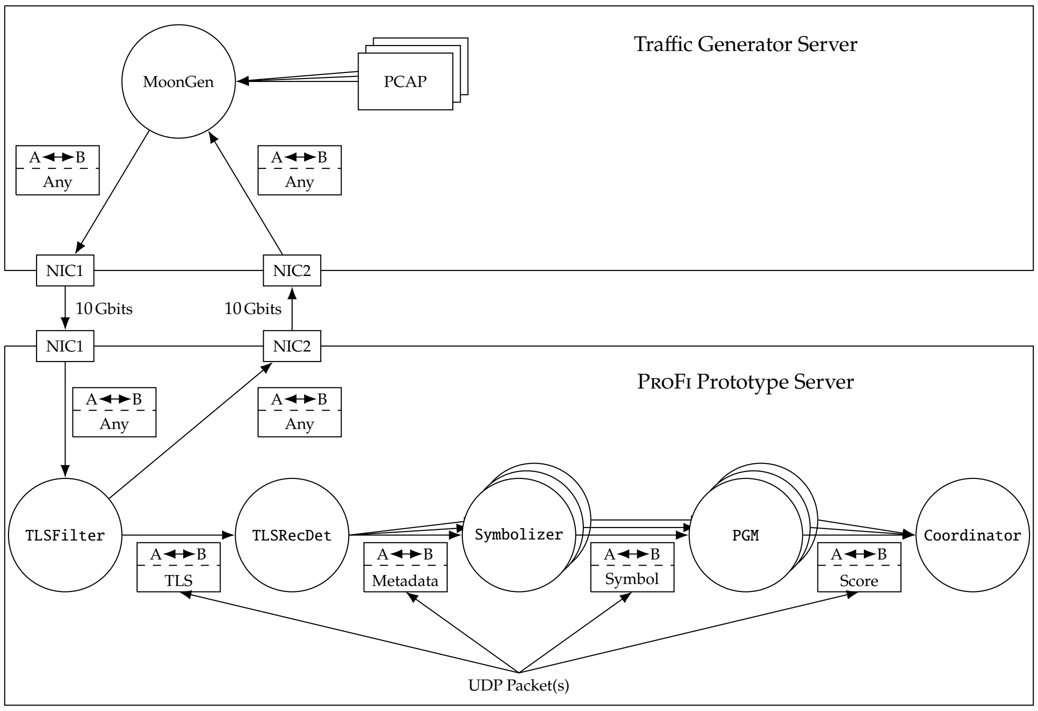


Figure 3.17 Testbed setup and prototype structure. Circles correspond to processes, i.e., VNFs. Rectangles to data sources and sinks. The prototype is implemented on top of `OpenNetVM`. VNFs in `OpenNetVM` exchange information via User Datagram Protocol (UDP) packets.

the scaling, addition, and updating of models with new parameters. For example, additional PGMs or symbolizers can easily be added by starting a new service. Already running services can be scaled as needed by deploying new instances.

`TLSFilter` Service. The `TLSFilter` separates TLS traffic from non-TLS traffic. For this purpose, the `TLSFilter` maintains an internal flow table that stores all currently active TLS connections. To filter traffic, the `TLSFilter` checks if, for every packet the `TLSFilter` receives from NIC1, the packet belongs to a TCP flow and has a payload. If not, the `TLSFilter` forwards the packet directly to NIC2, i.e., back into the network. If yes, the `TLSFilter` checks if the packet belongs to a TLS flow in its table. For this, the `TLSFilter` forms a symmetric key from the five tuple by sorting source IP, source port, protocol, destination IP, and destination port. This makes the key invariant to the packet's direction. If no entry exists, the `TLSFilter` checks if the payload contains a `client_hello`. If the payload contains a `client_hello`, the `TLSFilter` adds a rule to the table, copies the IP header and TCP payload into a UDP packet, forwards the UDP packet to the `TLSRecDet`, and forwards the original packet to NIC2. If the payload does not contain a `client_hello`, the `TLSRecDet` stops processing the packet and forwards it to NIC2.

If an entry exists, the `TLSFilter` copies the IP header and TCP payload into a UDP packet and forwards it to the `TLSRecDet`. If the maximum number of packets necessary to classify a flow on any PGM is reached, the TLSFilter removes the flow from its table. Subsequent packets are no longer duplicated, reducing the load on the TLSRecDet.

The TLSFilter detects TLS traffic based on the client_hello, which is one of the few messages in the TLS protocol in which the TLS header is guaranteed to start at the beginning of the TCP payload [128]. In the TLS record protocol, the TLS header can be at arbitrary positions due to fragmentation [128]. If a record spans multiple TCP segments, there might be no TLS header at all. In addition, fragmentation might place the TLS header into different packets. The first part of the header is located at the end of the payload of the first packet, and the remainder is located at the beginning of the second packet.

TLSRecDet Service. The TLSRecDet extracts TLS records from the payload in the UDP packets received from the TLSFilter. Since the prototype is evaluated with real traffic, the TLSRecDet must be able to handle events such as duplicate packets, re-transmissions, and varying TCP and IP header lengths. For this, the TLSRecDet maintains a flow table keyed with the symmetric five-tuple. For each flow, the TLSRecDet stores the five-tuple, the last time the flow table entry was matched, the number of times the flow table entry was matched, the next expected sequence number in each direction, and the outstanding bytes of the current TLS record for each direction. The TLSRecDet uses the sequence numbers to detect duplicate or missing packets. The TLSRecDet uses the outstanding bytes to retrieve the next TLS record header in the TCP payload in case the record spans multiple segments. After extracting the TLS records, the TLSRecDet overwrites the UDP packet's payload with the length of the TCP payload and information of the extracted TLS records. The TLSRecDet then forwards the UDP packet to the Symbolizer service.

The TLSRecDet iterates over the flow table and checks if a flow expired, i.e., no new packet has been received for a configurable amount of time. The TLSRecDet sends a UDP packet with a special ending symbol to the Symbolizer services for each expired flow. Then, the TLSRecDet removes the flow from the flow table. This ensures that every flow gets classified, even if less than the required number of packets arrived.

Symbolizer Service. The Symbolizer services map the extracted information into a symbol for a PGM. One Symbolizer service exists for every PGM. The Symbolizer overwrites the TLSRecDet packet's payload and forwards it to a PGM service. The symbol the Symbolizer service creates depends on the data preparation required for a PGM.

PGM Service The PGM service calculates an anomaly score for every TLS flow and forwards the score to the Coordinator. For each flow, the PGM service

Trace	Websites/s	TCP Flows/s	Packets/s [Mpps]
high-pps	147.71	46 551.04	2.24
high-fps	153.15	47 975.59	1.95
regular	424.17	16 446.72	1.65

Table 3.5 Trace statistics.

maintains the current state of the algorithm that calculates the log-likelihood. For the MC, the `PGM` service stores the previous symbol and a running sum of the log-likelihoods. For a PHMM, the `PGM` stores the variables for the forward algorithm. If the `PGM` received the configured number of symbols or the `TLSRecDet`'s flow-ending message, the `PGM` overwrites the payload of the UDP packet with the calculated anomaly score, removes the flow-table entry, and forwards the packet to the `Coordinator`. If the `PGM` expects more packets, the service drops the received UDP packet.

`Coordinator` Service. The `Coordinator` accumulates the anomaly scores of all `PGM` services for each flow and drops the received UDP packets. The `Coordinator` labels the flow with the `PGM` with the lowest anomaly score and writes the result to file.

3.7.2 Prototype Results

This section presents the network performance-related metrics. Specifically, this section investigates the processing cost of each service, the time the system needs to conclude on a label for a flow, and the number of `PGM` services that can be co-located on one CPU core.

Material and Methods

The prototype operates on a server with an Intel(R) Xeon(R) CPU E5-265 with 24 cores, 126 GB RAM, and an Intel(R) X540-AT2 network card with two 10 Gbit/s ports (Fig. 3.17). The server is connected back-to-back to a second server with identical hardware running MoonGen [170]. MoonGen replays Packet CApture Files (PCAPs) constructed from the traces collected according to Sec. 3.5.1, i.e., not with passively collected traces. MoonGen replays the PCAPs as fast as possible, i.e., at the full 10 Gbit/s. The prototype can use 17 out of the 24 cores for `PGM` and `Symbolizer` services. `OpenNetVM` blocks four cores and three cores are allocated to the `TLSFilter`, `TLSRecDet`, and `Coordinator`.

Two challenging workloads and one regular workload are created to benchmark the prototype. Tbl. 3.5 lists the workload characteristics. The regular workload consists of a random sample of webpage calls. The adversarial workloads *high-pps* and *high-fps* challenge the prototype with a higher packet and flow rate than the regular workload. This challenges the `TLSFilter`, `TLSRecDet`, and `PGM`

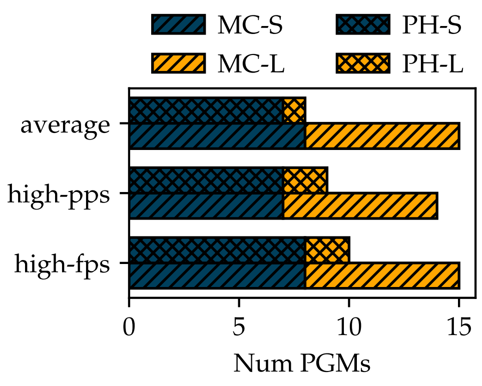


Figure 3.18 Number of PGMs on one CPU core. Suffixes -S/-L refer to small/large model.

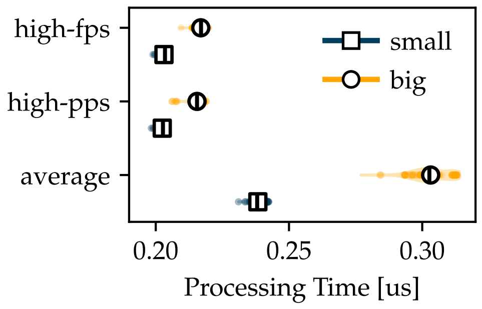


Figure 3.19 Distribution of average processing times of the `TLSFilter`.

services, respectively. Each trace has a volume of around 13 GB, resulting in a 10 s measurement on 10 G.

Number of classifiable websites

Fig. 3.18 shows the number of PGM services that can be co-located on one CPU core. A small and large MC and PHMM model is evaluated. The big PHMM model has 639 parameters. The small PHMM has 114 parameters. The MC models have 11 078 and 5 parameters respectively. The big model require 30 packets, the small models 5 packets for classification. For each PGM and workload, the number of PGMs is varied in $\{1, \ldots 10\}$. The highest number of co-located PGMs that could handle the traffic is reported.

Fig. 3.18 shows that between 1 and 8 PHMMs can be co-located on one CPU core. The values are similar across workloads. Fig. 3.18 shows that between 6 and 8 MCs can be co-located on one CPU core. A CPU core can fit more MCs than PHMMs, which is expected since the PHMM requires more computational resources to compute the log-likelihood of a sequence. Depending on the models' sizes, the prototype could run between 20 and 100 PGM services.

Impact on user traffic

Dataplane traffic traverses only the `TLSFilter` service. Fig. 3.19 illustrates the average processing times of the `TLSFilter` as violinplot. Each violin is computed

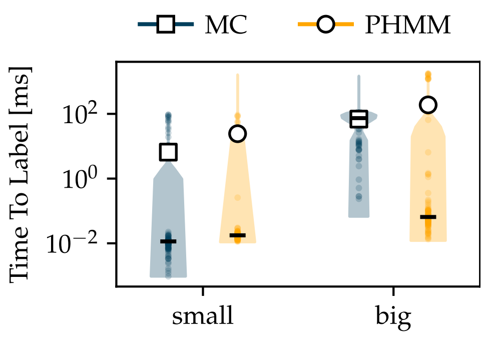


Figure 3.20 Time-to-label for small/large PGMs on the average workload.

from 20 runs. Fig. 3.19 shows that the `TLSFilter` is cheap and takes on average between 0.2 µs and 0.33 µs for each packet. Fig. 3.19 shows that the number of packets needed for classification affects the processing time of the `TLSFilter` function. This is expected since more packets result in a larger state of the `TLSFilter` function, which increases the processing time.

Time-to-label (T2L)

The Time to Label (T2L) is the duration between the last required packet entering the prototype and when the label is available [23]. For example, if 30 packets are required to classify a flow, then the T2L is the difference between the arrival time of the 30th packet and the time the `Coordinator` labels the flow. The arrival time of the first packet is not used because this would introduce noise. The T2L would depend not only on the prototype's latency but also on the inter-arrival time of packets in the trace.

The T2L for the small and big PGM variants is evaluated. The T2L with one PGM each is measured. Fig. 3.20 shows violin plots of the T2L. Fig. 3.20 shows a large variance in the T2L, ranging from 0.001 ms up to 1 000 ms. The median T2L for small models are at ~0.01 ms for PHMM and MC. The median values for the big PGMs vary. The big MC has a median T2L of ~100 ms, and the PHMM of ~0.06 ms. However, the distribution of T2Ls for the PHMM is heavy-tailed, resulting in an average of ~200 ms.

3.7.3 Summary

This section presents a prototype architecture that classifies traffic with PGMs and presents measurements obtained with real traffic traces. The prototype can hold 10 Gbits, corresponding to up to 424 website calls per second. Statistical traffic classification with PGMs is thus feasible in the network at line rate. The evaluations show that the prototype can monitor up to 100 websites. The results of this section have an impact beyond the WFP use-case since the ProFi could also label traffic for legitimate reasons.

At the same time, ProFi opens up new research opportunities in traffic classification. This chapter presents the first design of a microservice-based system able to perform WFP at the network edge. The investigation of how such systems, e.g., for statistical traffic classification, might be operated on top of a MEC infrastructure with computing facilities at different distances to the user could be an interesting avenue of future work. Further, the evaluation shows that the prototype's T2L has a high variance, potentially caused by the effect described in Chapter 5. Future work could focus on reducing the variance.

3.8 Ethical Considerations

Careful steps were taken to ensure that all captured data is in compliance with ethical standards. First, no personal data was collected during crawls of public and popular websites. Second, the crawling rate is limited to ensure that the websites are not impacted. The number of crawls is limited to 100 per day for websites within the Alexa Top 1 000 Ranking to minimize the impact of the study.

WFP demonstrates the ability of an adversary to launch WFP attacks at scale. This raises an ethical question, as censors can use published attacks to violate user privacy. To mitigate this effect, this chapter presents a defense that can readily be implemented alongside the attack. In addition, the preemptive exposure of potential vulnerabilities is necessary to develop effective and efficient countermeasures. The results in this chapter show the practicability and severity of WFP, sending an impulse to discuss the implementation of defense mechanisms not only in the Tor network, but also in common SSL libraries, and the TLS standard.

3.9 Conclusion

The encrypt-everything movement resulted in a substantial deployment of TLS, hiding accessed services and websites from legitimate and illegitimate observers. This chapter presents ProFi, a WFP attack on the flow level that de-anonymizes encrypted network traffic of websites that Internet users visit. To scale to large traffic aggregates, ProFi uses computationally cheap ML models and requires only the direction, packet size, and TLS record types of, at most, the first 30 packets of a flow. ProFi achieves a precision and recall of 86.51 % and 85.53 % in a closed-world, and 68.90 % and 78.71 % in an open-world scenario with 96 websites and 4 800 webpages. To show that ProFi could be deployed by SPs or other entities at scale, this chapter's results show that ProFi can operate as a middlebox in an online scenario and evaluate a prototype implementation in a testbed with real traffic traces. This chapter's results show that ProFi has the potential to process up to 424 websites/s at 10 G—a scale that is required to run the attack on the Internet, e.g., at an edge cloud, yet a scale to which current WFP attacks are not applicable. Further, the prototype shows that ProFi can label traffic within microseconds after receiving the last required packet, which could allow active interference with users' traffic. Yet, this chapter's results show that the attack can be mitigated by standard features available in TLS 1.3. To improve Internet privacy, such defense mechanisms should find their way into TLS libraries to protect user privacy on the Internet. Further, the analysis shows that standardizing the TLS handshake could help to improve privacy since ProFi frequently relied on characteristics of the handshake for classification.

At the same time, ProFi offers an opportunity to SPs to gain insights into the application and service distribution in their networks. This would empower SPs to make decisions about their network infrastructure to ensure the best possible service for their clients.

Future work on PGM-based WFP attacks could investigate the inclusion of auxiliary data into calculating the probability of a website to improve the classification performance. Further, the operation of the proposed microservice-based implementation on top of MEC infrastructure could be an interesting avenue for future work in statistical traffic classification.

Chapter 4

Data-driven Traffic Engineering Algorithm Design

Data center workloads are diverse and range from client-server, ML, and web traffic to high-performance computing applications [77, 171, 172]. Each workload has different requirements for Traffic Engineering (TE). For example, optimizing the completion time of individual flows is desirable for client-server applications [173] but decreases performance in ML workloads [174]. Each workload needs a TE policy tailored to its specific requirements to achieve the best performance. Indeed, new applications and network architectures regularly trigger the development of new TE policies [174–176]. To handle the generally bursty and unpredictable data center traffic [77, 171, 177, 178], recent proposals converged on distributed TE mechanisms that make forwarding decisions for individual flowlets [179] locally on network nodes based on the global network state [175, 180–182]. Optimizing TE to a datacenter's workload can thereby help to improve resource utilization and service quality, resulting in a competitive advantage [173]. For example, tuning the TE mechanisms in datacenters constituting a MEC infrastructure to the services and applications running there could improve the service quality and/or resource utilization, resulting in a competitive advantage for SPs. However, the distribution of applications in cloud locations might vary and change over time, making frequent readjustments of TE policies and TE mechanisms necessary.

Deriving a deployable distributed TE mechanism for a TE policy is challenging. A distributed TE mechanism includes the specification of update messages that disseminate the global network state in the network and the algorithms that compute forwarding decisions from the exchanged state [175]. Designing each component is challenging since the TE mechanism must react within milliseconds and cope with limited computational resources [175, 180]. The implementation must be efficient and holistic, making exploiting patterns in the network and the traffic necessary [175, 180, 182]. For instance, `HULA` [175] integrates a substantial part of the calculation for the forwarding decision into the probing of paths, essentially fusing the exchange of network state with the calculation of forwarding decisions.

To automate and simplify the translation of a TE policy to a distributed TE mechanism, this chapter presents MISTILL. MISTILL distills the forwarding behavior of a TE policy from exemplary forwarding decisions into a NN using ML. MISTILL learns (i) the content of update messages, i.e., a representation of switch local state suitable for transmission, (ii) which switches' update messages a switch needs to make a forwarding decision for a flowlet, and (iii) the computation of forwarding decisions from update messages. MISTILL removes the need to manually design update messages, information exchange, and the calculation of forwarding decisions. ML allows MISTILL to detect helpful patterns in traffic and network automatically and enables MISTILL to learn a distributed TE mechanism. Further, MISTILL's NA makes the learned patterns accessible to humans and can thus aid manual design processes. In contrast to previous work, MISTILL is not yet another TE mechanism aiming to outperform existing ones in scenarios they are optimized for. Instead, MISTILL is a method to automate the design of distributed TE mechanisms by learning them from data.

The main contributions in this chapter are:

1. A procedure to translate solutions of optimization problems for TE into learning problems.
2. An efficient way to train NNs representing a TE policy's behavior, exploiting the Markov Property present in many popular TE mechanisms.
3. The design of an ML model that learns forwarding behavior and makes learned patterns accessible to humans.
4. This chapter shows in simulations that MISTILL can learn distributed TE mechanisms for three TE policies that generalize to previously unseen traffic patterns.
5. This chapter analyzes the learned messages and their exchange.
6. A Proof-of-Concept (PoC) implementation based on the extended Berkeley Packet Filter (eBPF) - to the best of knowledge, the first implementation of a distributed NN-based TE mechanism on hardware.

The code to generate data and train and evaluate models is made publicly available[1]. This chapter shows that MISTILL can learn distributed TE mechanisms that closely implement the desired policies and generalize to previously unseen traffic patterns. This chapter analyzes the learned messages' content. This chapter shows that the NN learns an information exchange resembling an edge cover, which can be used to optimize the information exchange of other TE mechanisms. Finally, this chapter demonstrates with the PoC that MISTILL reacts at a millisecond scale to changes in the network and that MISTILL can operate in legacy networks.

[1]Code available at: `https://github.com/tum-lkn/mistill`

Content and outline of this chapter. The chapter is organized as follows. Sec. 4.1 introduces background information and related work. Sec. 4.2 describes the application scenarios and the requirements of TE towards ML models. Sec. 4.3 formalizes the notion of a TE policies and introduces four common policies for which MISTILL learns TE mechanisms. Further, Sec. 4.3 designs a learning system that translates solutions of TE policies to learning objectives, exploiting special properties in solutions of TE policies. Then, Sec. 4.3 describes the NA, its design choices, and specifies inputs and outputs. Sec. 4.4 presents and evaluates ML-friendly node addressing schemes. Sec. 4.5 evaluates the learned TE mechanisms in simulations focusing on ML-related performance indicators and visualizes the learned information exchange. Sec. 4.6 presents and evaluates the PoC implementation focusing on systems engineering-related performance indicators. Sec. 4.7 discusses the limitations of MISTILL in its present form, and Sec. 4.8 concludes the chapter. The sections Sec. 4.1.4, Sec. 4.2, Sec. 4.3.4-4.3.7, and Sec. 4.4-4.6.7 are based on previously published articles [30, 52].

4.1 Background and related work

This section introduces background information on TE in Sec. 4.1.1, and DCN topologies in Sec. 4.1.2. Then, Sec. 4.1.3 introduces background on a special NN architecture called Attention, and 4.1.4 introduces related work. This section reflects in part MaLANe's *Analyze Networked System* activity.

4.1.1 Traffic Engineering

SoA in TE for DCNs assign flowlets [179] to one of many paths from the flowlet's source to its destination [175, 180–182]. A flowlet is defined as a sequence of packets where each packet has an inter-arrival time smaller than a defined threshold. A normal flow identified by the five tuple can thus consist of multiple flowlets. Making forwarding decisions at the granularity of flowlets allows the re-assignment of a flow to another path while maintaining the packet sequence at the receiver, which is important for transport protocols such as TCP. For example, routing based on flowlets allows the re-assignment of a long-lived flow that sends bursts of data occasionally, e.g., in case of a persistent database connection, to a less utilized path for each new burst, improving the flow's performance indicators. Similarly, in case of congestion, packet loss might occur, triggering the creation of a new flowlet and the re-assignment of the flow from the congested path to another path [179].

A flowlet's path is chosen based on the current state of the complete network from a TE policy that optimizes a certain objective. Specifically, the TE policy specifies the neighbor to which a network node should forward a flowlet. Objectives of TE policies relate to use-case-dependent performance indicators. For

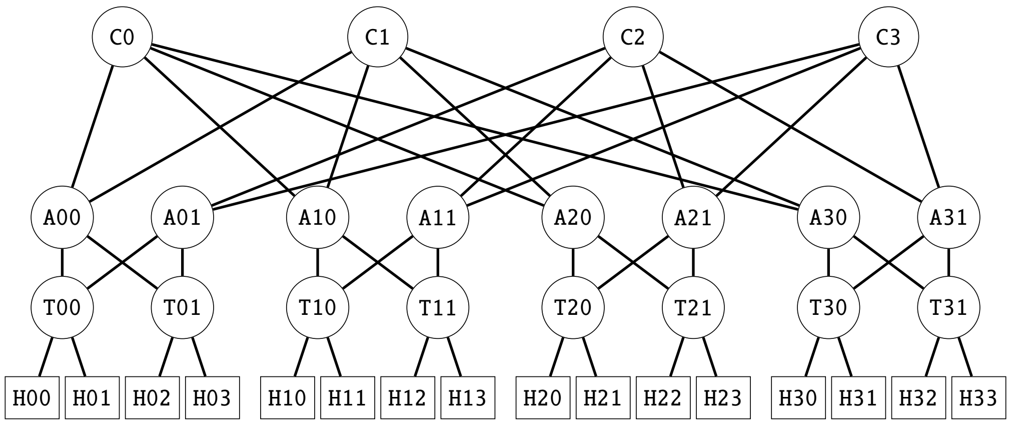


Figure 4.1 Fat Tree topology - A scalable three-tier Clos network. The topology is a multi-rooted tree with high path redundancy between any pair of hosts.

example, performance indicators used in the past are path length [182], path utilization [180], and latency [173].

A TE policy is realized in the network through a TE mechanism. The TE mechanism implements the necessary exchange of information and the processing of this information on the nodes such that the nodes can forward traffic according to the TE policy.

Since the design of a TE mechanism is challenging, new TE policies are often developed, assuming a centralized system as a first step. For example, using a packet or flow level simulation with an Integer Linear Program (ILP) or heuristic algorithm to compute forwarding decisions directly from the current state of the network. After the TE policy demonstrated its utility, a TE mechanism must be designed. Here, MISTILL hooks in.

4.1.2 Clos-Topologies

DCNs are a good fit for learning network protocols. Many data center topologies have regular and repetitive structures, and their addressing schemes can encode the location of network nodes (see Sec. 4.4). Clos networks are one such common data center topology. One special instance of Clos networks is the Fat Tree [183]; a topology with a three-tier structure that can interconnect large numbers of servers. Fig. 4.1 shows a $k = 4$ Fat Tree topology. A Fat Tree topology has three layers of switches: Top-of-the-Rack (ToR), aggregation, and core switches. Hosts are connected to ToR switches, which in turn are connected to aggregation switches. ToR and aggregation switches are organized in pods. The number k thereby specifies the number of pods and the number of ToR and aggregation switches in each pod corresponding to $\frac{k}{2}$ [183]. The Fat Tree in Fig. 4.1 has four pods interconnected by four core switches. A Fat Tree can support $\frac{k^3}{4}$ server nodes with $\frac{5k^2}{4}$ k-port switches. For instance, a Fat Tree constructed from 8-port switches can support 128 hosts with 80 switches. Clos networks provide a rich, high-capacity connectivity matrix, i.e., $\frac{k^2}{4}$ paths exist between any pair of

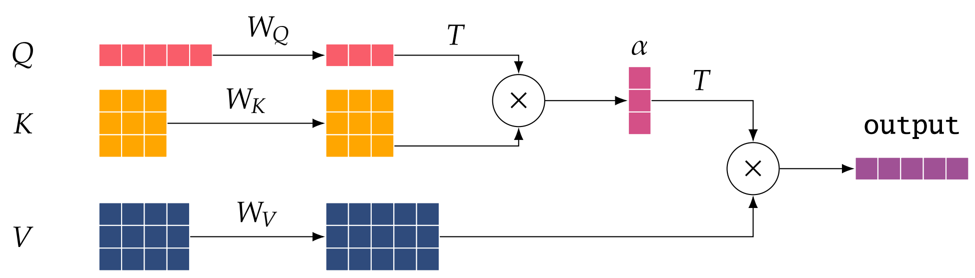


Figure 4.2 Illustration of the attention mechanism. The inputs `queries`, `keys`, and `values` are linearly transformed with corresponding weight matrices W_Q, W_K, and W_V. The inputs `keys` and `values` have the same number of rows. The transformed `queries` and `keys` are then multiplied to obtain the attention scores α representing a convex combination of the rows in the transformed `values`, which corresponds to the output layer of the attention mechanism. The transformed `queries` and the attention weights are transposed.

servers in different pods, and $\frac{k}{2}$ paths between servers in the same pod that are connected to different ToRs. The Fat Tree in Fig. 4.1 provides 4 paths between servers in different pods, and 2 paths between servers in the same pod connected to different ToRs.

Clos networks have two advantages: Simplified inventory management and resilience to failures [183, 184]. Inventory management is simplified because the network can be built from fixed-form switches. This makes it easy to stock spare devices and replace failed ones since each switch is the same. The resilience of the topology is increased through high path diversity. If a link or device fails, enough alternative routes exist.

4.1.3 Multi-Head Attention

MISTILL's NA uses Multi-Head Attention (`MHA`) [185] to learn which switches in the network contribute information to make a forwarding decision for a destination on a specific switch. `MHA` is heavily used in the context of Natural Language Processing (NLP) and behind many of the recent successes in the field [185]. `MHA` consists of multiple attention mechanisms, called attention heads, that are evaluated in parallel. Each attention head gets the same three inputs: Keys $K \in \mathbb{R}^{a \times b}$, Queries $Q \in \mathbb{R}^{c \times d}$, and Values $V \in \mathbb{R}^{a \times e}$, where $a, b, c, d, e \in \mathbb{N}$, i.e., each input corresponds to a matrix and K and V have the same number of rows. Each attention head computes for each query vector, i.e., row in Q, a convex combination of the rows in V conditioned on the query vector and the keys K. Fig. 4.2 illustrates this process.

To compute the convex combination, the three inputs are first linearly transformed using three weight matrices $W_Q \in \mathbb{R}^{d \times h_{QK}}$, $W_K \in \mathbb{R}^{b \times h_{QK}}$, and $W_V \in \mathbb{R}^{e \times h_V}$. The transformed keys and queries have the same number of columns h_{QK}. The values can have a different number of columns h_V. Each attention head has its weight matrices. The attention scores $A \in [0,1]^{a \times c}$ are then calculated as:

$$A = \mathrm{f}\left(\frac{1}{\sqrt{a}} K W_K \cdot (Q W_Q)^T\right). \tag{4.1}$$

The function f is applied along the columns normalizing the values such that:

$$A_{ij} \in [0,1] \qquad \forall 1 \leq i \leq a \wedge \forall 1 \leq j \leq c \tag{4.2}$$

$$\sum_{i=1}^{a} A_{ij} = 1 \qquad \forall 1 \leq j \leq c. \tag{4.3}$$

Each entry in A lies between zero and one, and the sum along each column results in one. Choices for f are the softmax, sparsemax and gumbelsoftmax functions. The final output of the attention head is then computed by:

$$A^T (V W_V). \tag{4.4}$$

The attention heads allow the NN to aggregate the rows in V differently. Specifically, the individual heads allow the NN to focus on different parts of the values. In the context of Mistill, one attention head could focus on aggregating network state from ToR switches in the destination pod, while another aggregates state of aggregation switches in the source pod, etc.

4.1.4 Related Work

Mistill is an interdisciplinary project and intersects with various research areas. This section reviews five aspects: Traditional routing and TE, systems engineering focusing on executing ML models inside networks, ML for routing and TE, and using ML-based performance models for flow scheduling. For each aspect, this section briefly introduces related work, then discusses the most related articles in detail, highlighting the differences to the present work.

Routing and Traffic Engineering

TE mechanisms for DCNs addressing different goals and challenges exist [186]. Solutions range from using Open Shortest Path First (OSPF) [187, 188] over new centralized designs [176, 189, 190] to distributed TE mechanisms that operate directly in the data plane [175, 180–182, 191]. In contrast to previous work, Mistill is *not* yet another TE mechanism and does not aim to outperform existing TE mechanisms in scenarios they are optimized for. Instead, Mistill is a method to automate the design of distributed protocols by distilling distributed TE mechanisms from data. In this line of research, `CONTRA` [182] is closest related to Mistill. Similar to Mistill, `CONTRA` allows the synthesis of TE mechanisms for different TE policies. However, `CONTRA` does not use ML but instead specifies a high-level language in which network operators can express their TE policy. CONTRA compiles the expressed policy language into P4 programs that implement the corresponding TE mechanism. The main novelty of Mistill is the use of ML to learn a TE mechanism for a TE policy from exemplary data. The research focuses on evaluating the learned TE mechanism's quality concerning the optimal policy, and the evaluation of the impact on network-level KPIs such as bandwidth overhead, reaction speed, and latency overhead.

In-Network ML

This area of related work has two aspects, explicit systems engineering for in-network inference of ML models [56–58, 192, 193] and learning of ML models that can replace or augment the Forward Information Base (FIB) of switches [194–196].

Concerning systems engineering, BaNaNa Split [57] is closest related to Mistill. BaNaNa Split investigates if programmable switches and SmartNICs can execute binarized CNNs to accelerate latency-sensitive ML workloads. Work in this chapter differs from BaNaNa Split in that the chapter investigates the learning of a TE mechanism for a TE policy with ML and the design of a potential implementation of such an ML-based TE mechanism. In contrast, BaNaNa Split does not train a specific NN but investigates if network equipment can run inference of a *given* binarized CNN. Thus, BaNaNa Split could complement the presented implementation in that the PoC's NN inference could move from the user space into kernel space and be offloaded onto a SmartNIC, assuming that the NN can be binarized without sacrificing too much accuracy. This approach could result in significant performance improvements.

Concerning FIB replacement, NuevoMatch [195] is closest related to Mistill. NuevoMatch learns an NN-based index structure for packet classification that can replace a switch's FIB. Like Mistill, NuevoMatch investigates how NNs can help cope with an extensive rule space. Specifically, NuevoMatch could compress rules that implement a sophisticated TE policy and accelerate the necessary packet classification. In contrast to Mistill, NuevoMatch takes the rule base as given. Further, NuevoMatch is implemented as a stand-alone VNF that runs on x86 hardware. In contrast, Mistill learns a distributed TE mechanism and proposes an implementation that can operate with legacy hardware.

ML and routing

Combinations of ML and networking have been explored from different angles in the past [197]. For routing, most related work considers (Deep) Reinforcement Learning ((D)RL) [198–211], while only a few works explicitly consider supervised learning [212–214]. Closest to Mistill is [212, 213], learning a distributed per-node forwarding policy. Geyer and Carle [212] and Xiao et al. [213] use Graph Neural

Network (GNN) to learn a distributed routing protocol akin to a distance vector algorithm.

Mistill differs from previous work for routing in its design towards practical deployability in the network and the realization in a PoC. Previous work does not consider the overhead of the NN beyond simple measurements of the NN inference times. Executing an NN in the data plane for each packet and flow is beyond current networking hardware. As the evaluations in this chapter show, the realization on hardware is a crucial aspect; just assembling the input to a NN can take longer than the actual execution. This chapter reports these effects for the first time, presents a deployable system runnable in legacy networks, and evaluates its performance on real hardware.

ML and performance prediction.

A recent line of work uses GNNs to predict various KPIs in networks [215–222]. Performance prediction is related to the task of Mistill since learning the forwarding behavior can be interpreted as learning the respective TE metric across neighbors. For example, the `MinMax` policy can be interpreted as learning the link utilization a flowlet would experience for each neighbor. Consequently, Rusek et al. [216] use their learned performance model to select suitable paths. However, in contrast to Rusek et al. [216], Mistill directly translates the network state into a forwarding decision. Performance models are not directly applicable to making forwarding decisions. Performance models evaluate different path candidates and are thus better suited to be used with a central controller that makes infrequent network configuration decisions. In contrast, Mistill targets a scenario where decisions must be made quickly and frequently in a distributed manner.

4.2 Application Scenario

This section introduces the operation and states the objective of Mistill high-level in Sec. 4.2.1, formulates the requirements towards the operation of a learned TE mechanism in DCNs in Sec. 4.2.2, and justifies the choice of using a NN to learn the mechanism in Sec. 4.2.3. This section corresponds mostly to MaLANe's *Translate Objective* activity.

4.2.1 The objective of Mistill

Mistill automates the design of a TE mechanism for a TE policy by learning a TE mechanism from exemplary forwarding decisions of a TE policy. Fig. 4.3 illustrates the process and expected outcome. Fig. 4.3 shows that Mistill uses the exemplary forwarding decisions from the centralized system to train a NN that represents a TE mechanism.

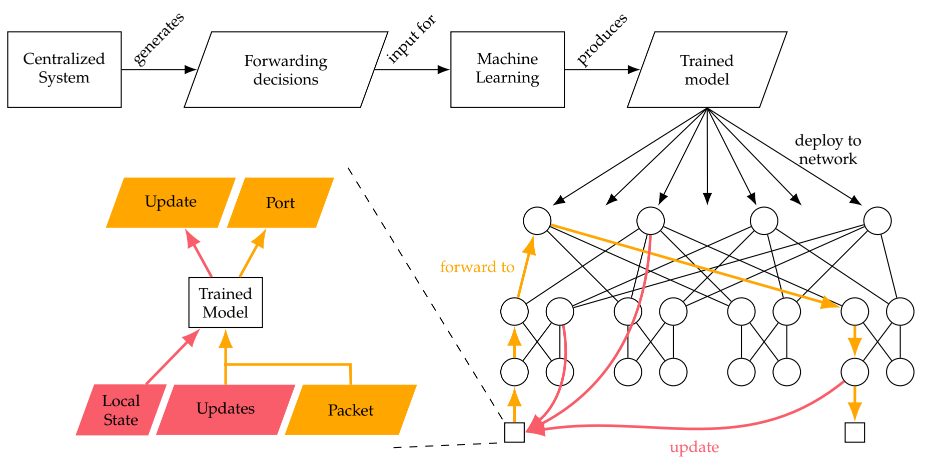


Figure 4.3 MISTILL trains a NN with a centralized TE policy's forwarding decisions and distributes the NN to hosts and switches for decentralized computation of updates and routes.

MISTILL uses the forwarding decisions to train an ML model, i.e., a NN. MISTILL trains the model offline, i.e., *not* on the network devices. The NN learns three aspects: 1) how to encode the local state in update messages, 2) how to exchange the update messages, and 3) how to map the exchanged state into forwarding decisions.

After training, MISTILL deploys the NN's parameters to the hosts and the switches. The switches use the NN to compute update messages from their local state (green in Fig. 4.3). The hosts use the NN to compute forwarding decisions from the switches' update messages and construct a path through the network (orange in Fig. 4.3). All hosts, resp. switches have *the same* NN parameters and use the NN in inference mode, i.e., do forward passes only.

The goal of this chapter is to design a NA that can learn a distributed TE mechanism for a TE policy, evaluate how well the NA learns the policy's behavior, investigate the impact of design choices, and show that the resulting TE mechanism can meet stringent performance requirements of DCNs.

Non-goals: MISTILL does not aim to improve upon existing TE mechanisms in their specific application scenarios. Instead, this work investigates the automated generation of TE mechanisms for novel TE policies with ML.

4.2.2 Data center network requirements

MISTILL's goal is learning distributed TE mechanisms for DCNs from exemplary data. This chapter focuses on Clos topologies due to their popularity and simple structure [223].

MISTILL learns distributed TE mechanisms since traffic in DCNs is bursty and unpredictable, requiring decision-making within milliseconds [77, 180]. Centralized controllers cannot handle events at such time scales. Their control loops run at the granularity of seconds [176, 189] and need tens of milliseconds to implement new routes [176]. Consequently, TE in DCNs moved to schemes that make local forwarding decisions based on the global network state [175, 180–182].

To make local decisions on the global network state, distributed TE mechanisms must exchange update messages at millisecond scale [175, 180, 182]. Thus, the updates must be small to keep the overhead low. To effectively utilize the high update frequency, forwarding decisions must be made regularly, e.g., for each new flowlet [179, 180]. This limits the computational demand of the decision logic, which, therefore, must be computationally efficient.

4.2.3 Why Neural Networks?

Mistill uses a NN to learn a distributed TE mechanism. A NN is a good ML model choice for three reasons.

1) NNs are general function approximators [224]. This is important because forwarding decisions can depend non-linearly on the network state. For example, minimizing the maximum link utilization results in forwarding decisions with non-linear dependencies. A small change in the utilization can lead to a sudden change in the forwarding decision. In contrast to other ML models such as Decision Trees, Gaussian Processes, or SVMs, NNs can be trained end-to-end[2]. This allows a holistic design and the learning of task-dependent intermediate representations, e.g., the content of update messages. 2) Inference with NNs uses simple arithmetic operations and is fast to perform, as Sec. 4.6 shows. 3) The execution of NNs is easy to parallelize, and accelerators for NN inference already exists for network equipment [58, 225] and are part of many CPUs and Field Programmable Gate Arrays (FPGAs) [226]. In addition, research efforts show that NNs can be executed in the data plane at line rate already on today's Application-specific Integrated Circuit (ASIC) [192], and end-hosts with SmartNICs [57]. An integration of Mistill with this type of hardware is thus likely to improve the results in Sec. 4.6.

4.3 Formal model and Neural Architecture Specification

Before presenting the NA to learn distributed TE mechanisms, this section first introduces a formal model of TE policies in Sec. 4.3.1, forming the basis of Mistill's learning task and showing Mistill's connection to classical optimization problems. Sec. 4.3.2 then describes how a TE policies solution translates to a learning objective and derives a learning objective optimizing NN parameters efficiently. Then, this section introduces the NA in Sec. 4.3.3, describes the NN inputs in Sec. 4.3.4, the computation of Hidden Node State Advertisement (`HNSA`)s

[2]End-to-end in this context refers to the ability of the NN to learn the content of update messages, which update messages are required to make forwarding decisions, and how to make the forwarding decisions in one architecture. An alternative approach could be to learn or design each aspect separately. End-to-end learning has the advantage that it can learn task-specific intermediate representations, e.g., tailor the update message's content to the specific TE policy.

in Sec. 4.3.5, the computation from which switches state is required in Sec. 4.3.6, and finally the computation of a forwarding decision in Sec. 4.3.7. This section largely corresponds to MaLANe's *Formalize Problem* activity.

4.3.1 Formal TE policy model

This section first introduces a graph representation for DCN topologies. Then, this section links TE policies to constraint optimization and formulates four TE policies as Constraint Satisfaction Problem (CSP) and Constraint OPTimization Problem (COPTP). The solution to the optimization problems then forms the basis for formulating MISTILL's learning task in the next section.

Graph Representation

Formally, a Fat Tree is modeled as a directed and attributed graph $G = (\mathcal{V}, \mathcal{E} \subset \mathcal{V} \times \mathcal{V})$. Links in the topology are modeled as directed edges in G. Thus, edges for both directions exist, i.e., $(u, v) \in \mathcal{E} \leftrightarrow (v, u) \in \mathcal{E}$. The function hosts : $2^{\mathcal{V}} \rightarrow 2^{\mathcal{V}}$ returns all hosts, the function tors : $2^{\mathcal{V}} \rightarrow 2^{\mathcal{V}}$ returns all ToR switches, aggs : $2^{\mathcal{V}} \rightarrow 2^{\mathcal{V}}$ all aggregation, and cores : $2^{\mathcal{V}} \rightarrow 2^{\mathcal{V}}$ all core switches. In addition, the functions ngh^- and ngh^+ return the successors and predecessors of a node $u \in \mathcal{V}$. Specifically the function

$$\mathrm{ngh}^- : \mathcal{V} \rightarrow 2^{\mathcal{V}}; u \mapsto \left\{v \in \mathcal{V} \,\middle|\, (u, v) \in \mathcal{E}\right\} \tag{4.5}$$

returns all nodes that are reachable from u. Similarly, the function

$$\mathrm{ngh}^+ : \mathcal{V} \rightarrow 2^{\mathcal{V}}; u \mapsto \left\{v \in \mathcal{V} \,\middle|\, (v, u) \in \mathcal{E}\right\} \tag{4.6}$$

returns all nodes from which u is reachable. Each edge $e \in \mathcal{E}$ is associated with (time-dependent) attributes such as the utilization, delay, etc., abstractly represented by a cost function:

$$\mathrm{c} : \mathcal{E} \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}. \tag{4.7}$$

Further, let the function:

$$\mathrm{up} : \mathcal{E} \times \mathbb{R}_{\geq 0} \rightarrow \{0, 1\} \tag{4.8}$$

return the availability of an edge $e \in \mathcal{E}$ at time t, and let G_t represents the values of all attributes in G at time t, i.e., the graph's state at time t. Further, let the function:

$$\mathrm{d_{min}} : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{N} \tag{4.9}$$

return the shortest path length between two nodes where each edge is associated with a cost of 1.

TE policies and Constraint Optimization

Routing a flowlet $f = (\varphi^+, \varphi^-, t)$ arriving at time t with source $\varphi^+ \in \text{hosts}(\mathcal{V})$ and destination $\varphi^- \in \text{hosts}(\mathcal{V})$ through G can be formulated as a CSP [227]. The CSP has for every edge $e \in \mathcal{E}$ a binary variable $x_e \in \{0, 1\}$, where $\mathcal{X} = \{x_e \mid e \in \mathcal{E}\}$ represents the set of flow variables, and $\mathcal{X}_e$ is equivalent to $x_e \in \mathcal{X}$. A value $x_e = 1$ indicates that edge e is part of f's path from φ^+ to φ^-. In addition, the CSP has the following constraints [227]:

$$\sum_{v \in \text{ngh}^-(\varphi^+)} x_{(\varphi^+, v)} = 1 \tag{4.10}$$

$$\sum_{v \in \text{ngh}^-(u)} x_{(u,v)} = \sum_{v \in \text{ngh}^+(u)} x_{(v,u)} \qquad \forall u \in \mathcal{V}/\{\varphi^+, \varphi^-\} \tag{4.11}$$

$$\sum_{v \in \text{ngh}^+(\varphi^-)} x_{(v,\varphi^-)} = 1 \tag{4.12}$$

$$x_e \in \{0, 1\} \qquad \forall e \in \mathcal{E}. \tag{4.13}$$

Constraint (4.10) specifies that the flowlet f has to leave the source node. Constraint (4.12) states that the flowlet f has to enter the sink node. Constraint (4.11) ensures that the flowlet f enters and leaves intermediate nodes, i.e., is not blackholed. Constraint (4.13) states that the flow variables are binary, i.e., can either be one or zero. Constraint (4.13) reflects that flowlets are unsplittable, i.e., packets belonging to a flowlet must use the same path through G.

The CSP in its present form is basic and returns any path from φ^+ to φ^- satisfying constraints (4.10)-(4.13) without considering the path's length, utilization, latency, and the availability of edges. A CSP can include edge attributes through additional constraints or by introducing an optimization objective, turning the CSP into a COPTP. Both, a CSP and a COPTP, can have multiple solutions $\boldsymbol{\mathcal{X}} = \{\mathcal{X}^1, \dots, \mathcal{X}^s\}$, where each element in $\boldsymbol{\mathcal{X}}$ corresponds to a set of flow variables. When referring to specific solutions, a superscript is used, i.e., $x_e^i \in \mathcal{X}^i$ refers to an edge's flow variable in the i^{th} solution. Further, every solution $\mathcal{X}$ can be represented through a path $\Omega \in 2^{\mathcal{V}}$, and the set $\boldsymbol{\Omega} = \{\Omega_1, \dots, \Omega s\}$ corresponds to the paths constructed from solutions $\mathcal{X}_1, \dots, \mathcal{X}_s \in \boldsymbol{\mathcal{X}}$. The nodes $u \in \Omega$ are ordered by their distance $\text{d}_{\text{min}}(u, \varphi^-)$ to the destination node.

This section introduces four TE policies: Equal Cost Multi-Pathing (`ECMP`), Weighted Cost Multi-Pathing (`WCMP`), Least Cost Path (`LCP`), and `MinMax`.

Equal Cost Multi-Pathing

`ECMP` assigns flows to one of multiple shortest paths using path length as metric [228]. In practice, `ECMP` hashes flows based on their five-tuple to one of the available paths [180, 229]. In this chapter, the `ECMP` policy obtains the set of paths for a flowlet $f = (\varphi^+, \varphi^-, t)$ by enumerating all solutions of a CSP with constraints (4.10)-(4.13), and the additional constraints:

$$x_e \leq \mathrm{up}(e, t) \qquad \forall e \in \mathcal{E} \tag{4.14}$$

$$\sum_{e \in \mathcal{E}} x_e \leq \mathrm{d}_{\min}(\varphi^+, \varphi^-). \tag{4.15}$$

Constraint (4.14) ensures that available edges are used, and Constraint (4.15) constraints the number of used edges.

Weighted Cost Multi-Pathing

`WCMP` uses the same CSP as `ECMP`. In contrast to `ECMP`, `WCMP` weights the available paths, e.g., by replicating specific ports as targets for the hash function [229]. Thus, flows are no longer equally hashed to available paths. Assuming an edge (u, v) lying on any path from φ^+ to φ^-, i.e., $\exists X \in \boldsymbol{X} : X_{(u,v)} = 1$. Then, MISTILL uses a splitting ratio for link (u, v) proportional to the maximum over the sum of the free down-stream capacity calculated for all paths that pass through v, defined as:

$$\begin{aligned}\mathrm{cfc} :& \mathcal{V} \times \mathcal{V} \times \mathcal{V} \to \mathbb{R}_+; (u, v, \varphi^-) \mapsto \max_{X \in \boldsymbol{X}}[X_{(u,v)}(\\ &\sum_{(u',v') \in \mathcal{E}} X_{(u',v')} \mathbb{I}(\mathrm{d}_{\min}(v', \varphi^-) < \mathrm{d}_{\min}(u, \varphi^-))(1 - \mathrm{c}((u', v'), t)))].\end{aligned} \tag{4.16}$$

In Eq. (4.16), c represents the links utilization. Thus, Eq. (4.16) sums the available capacity along a path given by a solution's flow variables. The indicator function in Eq. (4.16) removes the edges upstream of u. The flow variable $X_{u,v}$ in front of the sum results in zero if the edge (u, v) is not part of the solution. Intuitively, Eq. (4.16) distributes flowlets over paths such that the risk of meeting bottleneck links is reduced[3].

Least Cost Path

The `LCP` policy selects for a flowlet $f = (\varphi^+, \varphi^-, t)$ paths, such that the cost of the edges in the path is minimized:

$$\min \sum_{e \in \mathcal{E}} x_e \, \mathrm{c}(e, t), \tag{4.17}$$

subject to Constraints (4.10)-(4.15). MISTILL enumerates all solutions to the problem and selects one at random. Multiple solutions can arise if, e.g., `LCP` uses

[3]Note that Eq. (4.16) does not represent a solution to the max-flow-min-cut problem [229]. In a Fat Tree, the max-flow-min-cut corresponds to the value of the `MinMax` objective for all but the ToR switches, since only for ToR switches the minimum cut will contain more than one edge given Constraint (4.15). Thus, MISTILL sums over the available capacity in Eq. (4.16) to increase the diversity in objectives for the learning task.

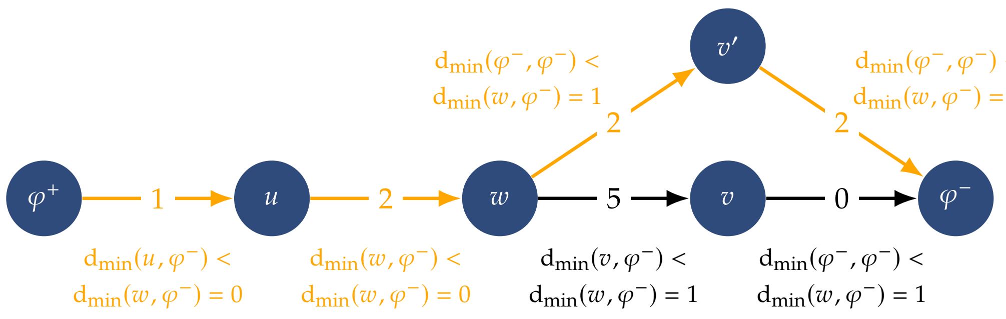

v'
φ^+
u
w
v
φ^-
1
2
5
0
2
2
$d_{min}(\varphi^-, \varphi^-) <$
$d_{min}(w, \varphi^-) = 1$
$d_{min}(\varphi^-, \varphi^-) <$
$d_{min}(w, \varphi^-) = 1$
$d_{min}(u, \varphi^-) <$
$d_{min}(w, \varphi^-) = 0$
$d_{min}(w, \varphi^-) <$
$d_{min}(w, \varphi^-) = 0$
$d_{min}(v, \varphi^-) <$
$d_{min}(w, \varphi^-) = 1$
$d_{min}(\varphi^-, \varphi^-) <$
$d_{min}(w, \varphi^-) = 1$

Figure 4.4 The figure shows two potential paths from φ^+ to φ^- that pass through the node w. Values on the edges represent the edge's cost. The equation next to the edges corresponds to the evaluation of the indicator function in the `MinMax` objective for node w. The optimal path is indicated with orange arcs.

queue lengths and the network is underutilized. Then, queues are unlikely to build up, and more than one path minimizes the cost. Constraint (4.15) limits the solutions to the set of shortest paths concerning path length, as usually done in Clos Topologies [175, 176, 180, 181, 189].

`LCP` is similar to `ECMP` and uses a time-dependent edge attribute to determine the shortest path from φ^+ to φ^- instead of a fixed weight.

MinMax

The `MinMax` policy selects for a flowlet $f = (\varphi^+, \varphi^-, t)$ paths, such that for each node u in the path from φ^+ to φ^- the maximum cost of the edges after u is minimized [175, 180]:

$$\min \max_{(u,v)\in\mathcal{E}} \left(x_{u,v}\mathbb{I}\left(d_{min}(v,\varphi^-) < d_{min}(w,\varphi^-)\right) c((u,v),t)\right) \qquad \forall w \in \mathcal{V}, \tag{4.18}$$

subject to Constraints (4.10)-(4.15), where Constraint (4.15) limits the solutions to the set of shortest paths wrt. path length. Eq. (4.18) represents multiple objectives, one for every node $w \in \mathcal{V}$. Intuitively, Constraints (4.10)-(4.15) result in a set of paths from φ^+ to φ^-. The individual objectives reduce these paths to those for which every subpath from any intermediate node to φ^- also minimizes the maximum link utilization. Fig. 4.4 illustrates the objective's effect for a node w. Fig. 4.4 shows two paths from φ^+ to φ^- that pass through w. Numbers on the arcs indicate the edge cost. The expression next to the arcs shows the evaluation of the indicator function in Eq. (4.18) for node w. Fig. 4.4 indicates the path optimizing Eq. (4.18) with orange color. Fig. 4.4 shows that the indicator function returns zero for all edges before w, i.e., their cost has no effect in the maximization in Eq. (4.18). Further, Fig. 4.4 shows that the indicator function returns one for the subpaths starting at w, i.e., Eq. (4.18) maximizes over their costs. The maximum link weight on subpath $s_1 = (w, v, \varphi^-)$ is five, and two on subpath $s_2 = (w, v', \varphi^-)$. Thus, the minimization in Eq. (4.18) sets the flow variables $x_{w,v}$ and x_{v,φ^-} to zero, and $x_{w,v'}$ and x_{v',φ^-} to one, i.e., selects subpath s_2.

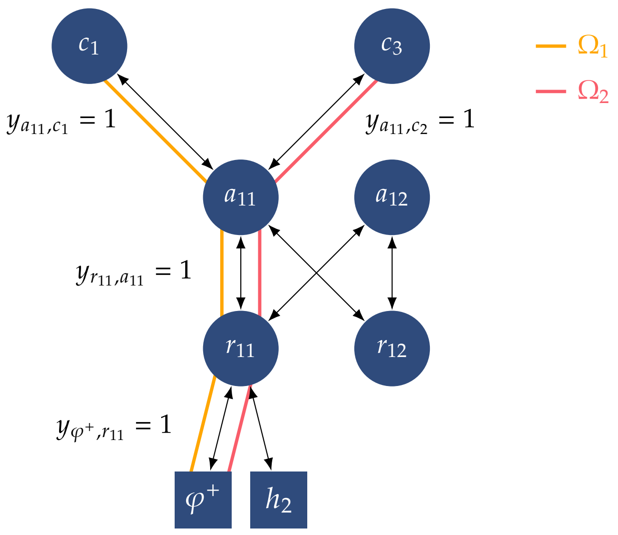


Figure 4.5 Shows the source pod of a flowlet f and illustrates a COPTP with two solutions using colored lines. Next to the edges of a solution, the figure shows the values of the target variables with a value of one for learning. Target variables that are not shown are set to zero.

The COPTP for `MinMax` can result in multiple solutions, which Mistill enumerates and selects from at random. Multiple solutions can arise if the link e that minimizes the maximum cost is close to φ^-. Then all paths that fulfill Constraint (4.15) and have smaller edge costs on edges before e are optimal solutions. Consequently, `MinMax`, as `LCP`, is a variant of `ECMP` that selects the paths based on time-varying costs on the edges.

Summary

A DCN topology is represented as an attributed graph G. The formulation of TE policies results in a set of flow variables x_e representing a path Ω through G from a source to a destination. The next section will use the flow variables to formulate a learning objective for Mistill.

4.3.2 Turning CSPs and COPTPs into learning problems

This section constructs a supervised learning task from the solution of CSPs and COPTPs. This section first describes a procedure that translates flow variables from solutions to a CSP or COPTP into learning targets and how the procedure handles infeasible problems. Then this section describes the underlying data model and derives optimization objectives that can be optimized with Stochastic Gradient Descent (SGD). This section then concludes with two algorithms describing how Mistill trains the NN's parameters and samples paths given a trained model.

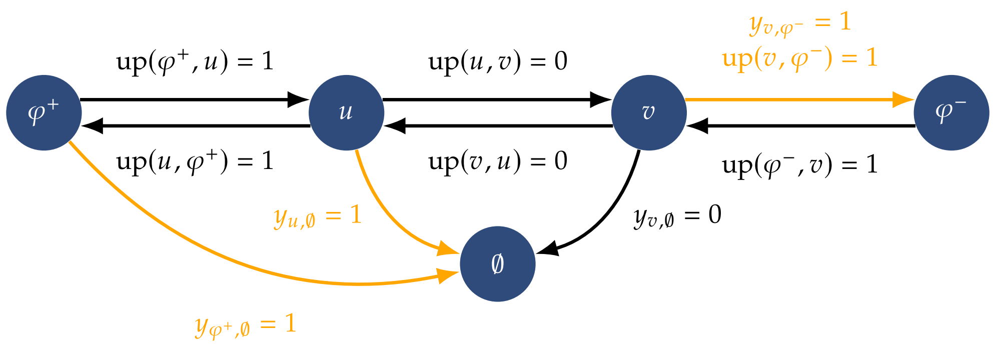

$y_{v,\varphi^-} = 1$
$\mathrm{up}(\varphi^+, u) = 1$
$\mathrm{up}(u, v) = 0$
$\mathrm{up}(v, \varphi^-) = 1$
φ^+
u
v
φ^-
$\mathrm{up}(u, \varphi^+) = 1$
$\mathrm{up}(v, u) = 0$
$\mathrm{up}(\varphi^-, v) = 1$
$y_{u,\emptyset} = 1$
$y_{v,\emptyset} = 0$
$\emptyset$
$y_{\varphi^+,\emptyset} = 1$

Figure 4.6 Illustrates the handling of infeasible problems. Special target variables are introduced, signaling that no path exists for a flowlet. Infeasibility is represented through the special node $\emptyset$ with respective target variables for each node in G.

Constructing targets

Mistill learns how a given node u forwards a flowlet $f = (\varphi^+, \varphi^-, t)$ to its neighbors given the cost and availability of edges at time t. Mistill constructs the training objective from target variables $\{y_{u,v} \in [0, 1] \mid v \in \mathrm{ngh}^-(u)\}$, where $y_{u,v} > 0$ if any flow variable $x^i_{u,v} = 1$ is set in a solution for a CSP or COPTP. For `WCMP`, $y_{u,v}$ is set to the normalized weight, i.e.:

$$y_{u,v} = \frac{\mathrm{cfc}(u, v, \varphi^-)}{\sum_{v' \in \mathrm{ngh}^-(u)} \mathrm{cfc}(u, v', \varphi^-)}. \tag{4.19}$$

Fig. 4.5 illustrates how the target variables are obtained for `ECMP`, `LCP`, and `MinMax`. Fig. 4.5 shows a subgraph of a $k = 4$ Fat Tree. The subgraph consists of the pod in which f originates and two core switches. Fig. 4.5 illustrates two solutions Ω_1, and Ω_2 using colored edges. Fig. 4.5 shows that the solutions overlap initially and then get forwarded to different core switches. Further, Fig. 4.5 shows the target variables set to one. Fig. 4.5 shows that the target variables $y_{\varphi^+,r_{11}}$ and $y_{r_{11},a_{11}}$ are one, since the respective flow variables for edges (φ^+, r_{11}) and (r_{11}, a_{11}) are set to one in both solutions. Similarly, the target variables y_{a_{11},c_1} and y_{a_{11},c_3} are both one since the corresponding flow variables $x^1_{a_{11},c_1}$ and $x^2_{a_{11},c_3}$ are one in either Ω_1 or Ω_2. Thus, the targets for Mistill for, e.g., node r_{11} and a_{11} are $y_{r_{11},\varphi^-} = 0$, $y_{r_{11},h_1} = 0$, $y_{r_{11},a_{11}} = 1$, and $y_{r_{11},a_{12}} = 0$, and $y_{a_{11},r_{11}} = 0$, $y_{a_{11},r_{12}} = 0$, $y_{a_{11},c_1} = 1$, and $y_{a_{11},c_3} = 1$.

Handling infeasible problems

To handle infeasible problems, Mistill uses the target variables

$$y_{u,\emptyset} = \mathbb{I}\left(\sum_{v \in \mathrm{ngh}^-(u)} y_{u,v} = 0\right), \tag{4.20}$$

defined for every node $u \in \mathcal{V}$. Fig. 4.6 illustrates how $y_{u,\emptyset}$ depends on the values of the other target variables. Fig. 4.6 shows a line graph in which the edges (u, v) and (v, u) are not available. Thus, no route exists to φ^- from φ^+ and u. Consequently, Fig. 4.6 shows that the target variables $y_{\varphi^+,\emptyset}$ and $y_{u,\emptyset}$ are set to one. In contrast, Fig. 4.6 shows that a solution v to φ^- exists. Thus $y_{v,\emptyset}$ is set to zero and y_{v,φ^-} to one[4].

Deriving learning objectives

To model the forwarding behavior for `LCP`, `ECMP`, and `MinMax`, Mistill uses independent Bernoulli distributions for each neighbor, i.e.:

$$y_{u,v} \sim \text{Bernoulli}(\text{nn}_{\phi_{b,v}}(u, \varphi^-, G_t)), \tag{4.21}$$

where $\text{nn}_{\phi_{b,v}}$ is a function parameterized with parameters $\phi_{b,v}$ modeling the probability of success, i.e., represents the Bernoulli distribution's probability of success. Thus, Mistill can predict the value for each target variable individually.

In the case where the forwarding policy can be represented as a distribution over neighbors as for `WCMP`, the forwarding behavior is modeled as Categorical distribution:

$$y_{u,v_1}, \ldots, y_{u,d_u}, y_{u,\emptyset} \sim \text{Categorical}(\text{nn}_{\phi_c}(u, \varphi^-, G_t)), \tag{4.22}$$

where $d_u = |\ \text{ngh}^-(u)\ |$, and nn_{ϕ_c} is a function parameterized with parameters ϕ_c modeling the Categorical distribution's event probabilities. In contrast to the Bernoulli distribution, a draw from the Categorical distribution results in an assignment for all target variables of a node u.

To efficiently learn the parameters $\phi_{b,v}$ and ϕ_c, Mistill exploits the fact that paths returned by a TE policy in Fat Tree DCNs can be interpreted as to fulfill the Markov Condition [80]. Taking on a probabilistic perspective, the probability of choosing a neighbor of switch u for a flowlet $f = (\varphi^+, \varphi^-, t)$ on any node v in graph G depends only on φ^- and the availability and cost of edges at time t, i.e., G_t. The independence arises from the property of shortest paths that any subpath of a shortest path is also a shortest path, which forms the basis of Dijkstra's Algorithm [227]. For `MinMax`, as formulated in Eq. (4.18), the property also holds since the next hops depend only on the down-stream link costs[5].

Thus, for `ECMP`, `LCP`, and `MinMax`, the joint probability of observing the target variables $y_{u,v_1}, y_{u,v_2}, \ldots y_{u,v_{d_u}}, y_{u,\emptyset}$ for nodes $u \in \Omega_f$ in a path for flowlet f, where

[4]Note that this procedure requires a separate check for every node u if no route to φ^- exists.

[5]Minimizing the maximum link cost with the objective $\min \max_{e \in \mathcal{E}} x_e c(e, t)$ would make the forwarding decision on a node u additionally dependent on the maximum link weight on the path from φ^+ to u. This is because if the maximum link weight is upstream of u, then all neighbors on a path fulfilling the length constraint are possible next hops since they, by definition, have a smaller link weight and are thus part of the solution.

$d_u = \mid \mathrm{ngh}^{-}(u) \mid$, can be modeled as d_u independent draws from a Bernoulli distribution, one draw for each neighbor [27]:

$$\prod_{u \in \Omega_f} \mathrm{nn}_{\phi_b}(y_{u,v_1}, \ldots, y_{u,\emptyset} \mid \varphi^{-}, u, G_t) = \prod_{u \in \Omega_f} \prod_{v \in \mathrm{ngh}^{-}(u) \cup \{\emptyset\}} \big(\mathrm{nn}_{\phi_{b,v}}(y_{u,v} \mid \varphi^{-}, u, G_t)^{y_{u,v}} \cdot (1 - \mathrm{nn}_{\phi_{b,v}}(y_{u,v} \mid \varphi^{-}, u, G_t))^{1 - y_{u,v}}\big). \tag{4.23}$$

For WCMP, the target variables are modeled as one draw from a Categorical distribution [27]:

$$\prod_{u \in \Omega_f} \mathrm{nn}_{\phi_c}(y_{u,v_1}, \ldots, y_{u,\emptyset} \mid \varphi^{-}, u, G_t) = \prod_{u \in \Omega_f} \prod_{v \in \mathrm{ngh}^{-}(u) \cup \{\emptyset\}} \mathrm{nn}_{\phi_c}(y_{u,v} \mid \varphi^{-}, u, G_t)^{y_{u,v}}. \tag{4.24}$$

Mistill optimizes the function's parameters ϕ by maximizing the probability of the target variables constructed from the solution of flowlet f for the nodes in one of the solution's paths Ω. Mistill maximizes the probability in the usual way by taking the gradient wrt. ϕ of the negative log-likelihood [27]. The log-likelihood for the Bernoulli distribution is given by [27]:

$$\begin{aligned}\mathcal{L}_{Ber} \doteq -\sum_{u \in \Omega} \sum_{v \in \mathrm{ngh}^{-}(u) \cup \{\emptyset\}} &\big(y_{u,v} \log \mathrm{nn}_{\phi_{b,v}}(y_{u,v} \mid \varphi^{-}, u, G_t) \\ &+ (1 - y_{u,v})(1 - \log \mathrm{nn}_{\phi_{b,v}}(y_{u,v} \mid \varphi^{-}, u, G_t))\big),\end{aligned} \tag{4.25}$$

the log-likelihood for the Categorical distribution is given by [27]:

$$\mathcal{L}_{Cat} \doteq -\sum_{u \in \Omega} \sum_{v \in \mathrm{ngh}^{-}(u) \cup \{\emptyset\}} y_{u,v} \log \mathrm{nn}_{\phi_c}(y_{u,v} \mid \varphi^{-}, u, G_t). \tag{4.26}$$

To learn a TE policy, a set of exemplary paths is generated with one of the policies. The loss functions $\mathcal{L}_{Ber}$ and $\mathcal{L}_{Cat}$ are used with SGD to optimize the NN's parameters computing the parameters for the Bernoulli and Categorical distributions.

Calculating the target variables per node has two advantages: 1) the NN's output layer corresponds to the maximum number of successors, i.e., maximum out-degree, of nodes in a DCN's topology, and 2) predicting the next hop improves data efficiency. Instead of using a node's neighbors as targets, the values of all flow variables or the available paths could be used instead. However, the number of edges and paths are large compared to the maximum degree. In a Fat Tree, each switch has k neighbors, compared to $\frac{k^5}{16}$ shortest paths, and $\frac{3k^3}{2}$ directed edges. Thus, in a $k = 8$ Fat Tree, each switch has 8 neighbors compared to 2 048 paths and 768 directed edges.

Predicting the next hop further increases the data efficiency of Mistill since many flowlets could have the destination φ^{-} and traverse node u. Since TE policies usually fulfill the Markov Property, the next hop would be chosen identically for all flowlets that arrive at u and have the destination φ^{-}. A single solution is thus representative of all other flowlets as well. With flow variables or paths as targets, Mistill could not benefit as strongly from this effect since the output would necessarily differ.

Learning parameters and sampling paths

This section introduces high-level algorithms for the parameter training and sampling of paths. Algorithm 1 describes Mistill's training procedure, and Algorithm 2 describes how Mistill samples paths from optimized parameters.

Algorithm 1: Mistill's training procedure.

```
Input: Graph $G$
repeat
    $\varphi^+, \varphi^-, t \leftarrow$ sampleFlowlet();
    $\mathcal{X} \leftarrow$ solveProblem($(\varphi^+, \varphi^-, t), G$);
    $\Omega \leftarrow$ getPath($\mathcal{X}$);
    $\mathcal{Y} \leftarrow$ makeTargets($\mathcal{X}, G$);
    Take gradient step on $\nabla_{\phi_b}\mathcal{L}_{Ber}$ or $\nabla_{\phi_c}\mathcal{L}_{Cat}$;
until converged;
```

Training. To train the parameters ϕ_b and ϕ_c, Algorithm 1 samples a flowlet. The function sampleFlowlet in Line 2 could, e.g., correspond to a packet or flow-level simulation, or, as in this chapter, sample source and destination pairs and edge attributes randomly.

Line 3 computes the solutions for a specific TE policy for f. Then, Line 4 chooses one path from the set of solutions, and Line 5 obtains the target variables. Finally, Line 6 computes the gradient for the appropriate loss function using $\mathcal{Y}$ and Ω.

Algorithm 1 repeats these steps until convergence.

Algorithm 2: Mistill's path computation procedure.

```
Input: Flowlet $f = (\varphi^+, \varphi^-, t)$, Graph $G$, optimized parameters $\phi$.
$u \leftarrow \varphi^+$;
$\Omega \leftarrow \emptyset$;
repeat
    $u \leftarrow$ sampleNeighbor($u, \varphi^-, G_t, \phi$);
    if $u = \emptyset$ then
        $\Omega \leftarrow \emptyset$;
    else
        $\Omega \cup \{u\}$;
until $u = \varphi^- \lor u = \emptyset$;
return $\Omega$
```

Sampling. To obtain a path after optimizing parameters ϕ_b or ϕ_c, Algorithm 2 iteratively samples a next hop from the learned distribution. Algorithm 2 ini-

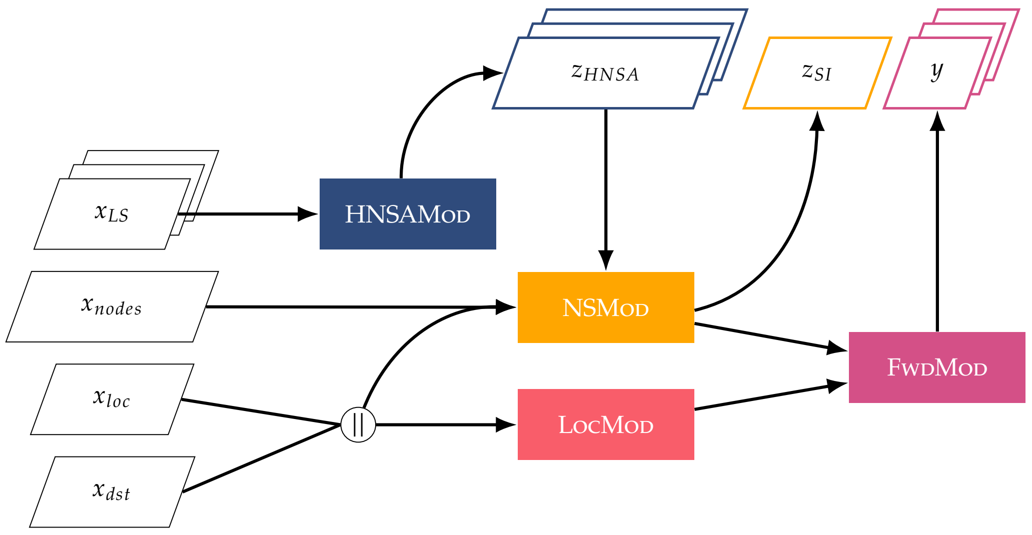


Figure 4.7 The NN architecture consists of four modules: The LocMod, the HNSAMod, the NSMod and the FwdMod.

tializes the next hop to the flowlet's source φ^+, and the path Ω to the empty set. Then, Algorithm 2 repeatedly samples the next neighbor. Each time Algorithm 2 samples a new neighbor, the algorithm checks if the neighbor corresponds to $\emptyset$, i.e., there is no solution to the corresponding problem. If $u = \emptyset$, then the algorithm sets Ω to the empty set and terminates. Else, Algorithm 2 appends u to Ω. Algorithm 2 repeats these steps until the sampled neighbor corresponds to φ^-. The function sampleNeighbor in Line 4 thereby uses the learned parameters to sample from the resulting Bernoulli or Categorical distribution.

In practice, Algorithm 2 can be optimized, especially for Fat Tree topologies. For example, u can be initialized to the ToR switch φ^+ connects to. Further, Algorithm 2 can stop once u corresponds to a Core switch since only one path exists from each Core to destination φ^- in a Fat Tree.

Summary

Mistill combines the flow variables $X_e^i \in \mathcal{X}$ from multiple solutions of an edge into a target variable y_e. Depending on the TE policy, the target variables are binary or continuous and modeled as independent draws from a Bernoulli and Categorical distribution, respectively. Parameterized functions represent the distribution's parameters, and the task of Mistill is to adjust the function's parameters such that the probability of the given target variables is maximized. Mistill maximizes the probability with SGD using the log-likelihood of the target variables. The next section introduces the parameterization, i.e., the NA computing the distribution's parameters.

4.3.3 Neural Architecture

Mistill uses a NN to compute the parameters for the Bernoulli and Categorical distributions modeling the target variables. The NN takes as input the flowlet's destination φ^-, a representation of node u, and the state of the graph G_{t_f}. In addition to learning the distribution's parameterization, Mistill's NA must also learn update messages that later inform hosts about the switches' local state.

The proposed NA consists of four modules: the Localization Module (LocMod), the `HNSA`[6] Module (HNSAMod), the Network State Module (NSMod), and the Forwarding Module (FwdMod). Fig. 4.7 shows how the modules interact and represent inputs and outputs with hashes. Mistill uses the same NN for all nodes, i.e., shares parameters. This section introduces the NA's inputs and outputs high-level. Subsequent sections will define them precisely.

Outputs. The NA has three outputs: `Ports` $y \in [0, 1]^{\hat{d}+1}$, `HNSA` content $z_{HNSA} \in \{0, 1\}^{d_1}$, and State Interest (`SI`) $z_{SI} \in \{0, 1\}^{d_2}$, where $d_1, d_2 \in \mathbb{N}$. `Ports` represents the target variables $y_{u,v}$ for the neighbors $v \in \text{ngh}^-(u) \cup \{\emptyset\}$, i.e., which outgoing edges are part of a solution to a CSP or COPTP, and thus through which port switch u should forward flowlet f. The `HNSA`s correspond to the update messages that inform hosts about the local state of the sending device. The `SI` signals which devices need to exchange information and corresponds to max-pooled attention scores.

Inputs. The NA has four inputs: `LinkState` $x_{LS} \in \mathbb{R}^6$, `AllNodes` $x_{nodes} \in \{0, 1\}^{|\mathcal{V}/\text{hosts}(G)| \times 24}$, `Destination` $x_{dst} \in \{0, 1\}^{24}$ and `Location` $x_{loc} \in \{0, 1\}^{24}$. `LinkState` is the switch local state, e.g., the availability and utilization of incident links. The NA receives the link state of all switches as input and transforms each switch's `LinkState` into a `HNSA`. `Location` is the identifier of a switch, `Destination` the identifier of a flowlet's destination φ^-, and `AllNodes` the identifiers of all switches. The NA uses `Location`, `Destination`, and `AllNodes` to learn how to condition forwarding decisions on a specific switch and destination and which switches' `HNSA`s are needed to make a forwarding decision.

Training. A NN is trained end-to-end. Training end-to-end tailors `HNSA`s and `SI` to a forwarding policy. The loss function depends on the TE policy. Policies, for which nodes can have multiple candidates to which they forward traffic to, are trained with the loss $\mathcal{L}_{Ber}$. TE policies that have only one neighbor, or, as `WCMP`, result in splitting weights are trained with $\mathcal{L}_{Cat}$ instead.

Changes in the network that lead to changes in the forwarding policy require a re-training of the model. For example, new networking hardware or changes

[6]In analogy to Link State Advertisement (LSA) in link-state routing protocols. *Hidden* refers to the fact that the content of the messages is the activations of a hidden layer of the NN. Further, `HNSA`s differ from LSAs because they are not per link but per node.

to the physical network topology. As this chapter will show, re-training is short compared to the deployment process of new hardware and can be integrated into the roll-out process.

Inference. The trained NN is deployed to switches and hosts. The switches use the NN to compute `HNSA`s from their local state. The hosts use the NN to select necessary `HNSA`s and calculate forwarding decisions. Hosts can calculate forwarding decisions by sampling from the resulting Bernoulli and Categorical distributions or by taking the maximum over returned parameters, i.e., taking the neighbor for which the NN indicates the highest confidence that it is part of the solution. The hosts and switches do not exchange the weights of the NN. The hosts and switches also never execute the full NN. To compute `HNSA`s, the switches execute the HNSAMod. To compute a switches' forwarding decision, the hosts execute the LocMod, NSMod, and FwdMod, but not the HNSAMod. The hosts use the `HNSA`s they received from the switches instead.

4.3.4 Data Format

`Location` $x_{loc} \in \{0,1\}^{24}$, `Destination` $x_{dst} \in \{0,1\}^{24}$, and `AllNodes` $x_{nodes} \in \{0,1\}^{|\mathcal{V}/\text{hosts}(\mathcal{V})| \times 24}$ are binary vectors and correspond to the last three octets of IP addresses following the addressing scheme for Fat Trees [183]. This chapter explores different node representations besides IP addresses and presents the results in Section 4.4.

Each `LinkState` $x_{LS} \in \mathbb{R}^6$ of an edge $(u,v) \in \mathcal{E}$ at time t is a real-valued vector obtained with the function:

$$\text{ls} : \mathcal{V} \times \mathcal{V} \times \mathbb{R}^+ \to \mathbb{R}^6; (u,v,t) \mapsto \begin{bmatrix} \text{up}((u,v),t) \\ 1 - \text{up}((u,v),t) \\ \text{up}((v,u),t) \\ 1 - \text{up}((v,u),t) \\ \text{c}((u,v),t) \\ \text{c}((v,u),t) \end{bmatrix}^T . \tag{4.27}$$

The first four attributes of a `LinkState` are a one-hot encoding of the link availability in each direction. For example, `10 10` means that both directions are up, and `10 01` means that one direction is up and one is down. The fifth and sixth attributes are the weight of the link in each direction, representing, e.g., link utilization.

In this chapter, the definition of ls uses the edge availability, and a numeric attribute since TE policies are usually constraint or optimized for only one such attribute [175, 180–182, 230]. To accommodate TE policies with multiple objectives, the corresponding attributes can be included in the definition of ls.

4.3.5 Learning the HNSA content

The HNSAMod transforms the `LinkState` of a switch into a binary vector $z_{HNSA} \in \{0,1\}^{d_1}$, where $d_1 \in \mathbb{N}$. The binary vector z_{HNSA} is the content of a `HNSA`. The vector corresponds to a hidden layer with binary activations. Since switches send the `HNSA` over the network, binary activations are better than real-valued activations. A binary representation allows the NN to have more neurons for the same number of bits. For example, a binary vector of 128 Bits corresponds to four single precision floating point numbers. Since NNs store information in a distributed representation, having more neurons with low precision is better than having fewer neurons with high precision [224].

The HNSAMod samples the binary activations from m independent, one-hot encoded categorical distributions of arity n:

$$z_{HNSA} \sim \text{Categorical}(n, \theta_1) \mid\mid \ldots \mid\mid \text{Categorical}(n, \theta_m). \tag{4.28}$$

The NN calculates the parameters θ_i by concatenating the link states of a switch $u \in \mathcal{V}$ to a node state $x_{NS} \in \mathbb{R}^{6 \cdot \hat{d}}$, i.e., the node state is defined as the return value of a function:

$$\text{ns} : \mathcal{V} \times \mathbb{R}^+ \to \mathbb{R}^{6 \cdot \hat{d}}; (u, t) \mapsto \mid\mid_{v \in \text{ngh}^-(u)} \text{ls}(u, v, t), \tag{4.29}$$

where $\hat{d}$ is the maximum degree of switches in the network. Vectors of switches with a smaller degree are zero-padded to this length. Then, the HNSAMod passes the node state through a feed-forward NN with one hidden layer and ReLU activation. Since TE policies operate on the current network state, i.e., not on past network states, a feed-forward NN is sufficient to encode the local state.

To keep the NN differentiable, a reparameterization trick is used to sample from the categorical distributions [231, 232]. During training, the NN learns how to parameterize the categorical distributions from a switch's link states to encode information in the binary hidden layer. The arity n and number of distributions m are hyperparameters of the NN architecture and optimized during training.

4.3.6 Learning to Communicate

The NSMod learns: 1) how to process `HNSAs` , and 2) the `SI` output $z_{SI} \in [0,1]^{d_2 \times |\mathcal{V}/\text{hosts}(\mathcal{V})|}$ with $d_2 \in \mathbb{N}$, i.e., which `HNSAs` are needed to compute forwarding decisions for a specific switch and destination.

To learn the `SI` and process `HNSAs` , the NA uses an attention mechanism inspired by the scaled dot-product attention [185][7], which can be interpreted as a learned lookup table that retrieves `HNSAs` based on a switch $u \in \mathcal{V}/\text{hosts}(\mathcal{V})$ and a destination $\varphi^- \in \text{hosts}(\mathcal{V})$[8].

[7]See Sec. 4.1 for background information on `MHA`.

[8]See Sec. 4.1.3 for background information on the attention mechanism.

For Mistill, the Queries Q corresponds to the concatenation of `Destination` and `Location` $Q = x_{dst} \mid\mid x_{loc}$, K to `AllNodes` x_{nodes} and V to the row-stacked `HNSA`s $z^{G}_{NHSA} := (z^{u_1}_{HNSA}; \ldots ; z^{|\mathcal{V}/\text{hosts}(\mathcal{V})|}_{HNSA})$, where $z^{u_i}_{HNSA}$ identifies the `HNSA` of the i^th switch. The NA use `MHA`, i.e., d_2 parallel attention mechanisms, and concatenates their output. The concatenated output is passed through a hidden layer with ReLU activation and then into the FwdMod.

The `SI` corresponds to the attention scores of the individual attention heads. Non-zero entries signal that the corresponding `HNSA` contributes information for the forwarding decision. The scores A make learned patterns explicit and interpretable. To enforce sparse patterns, the NA calculates attention scores with the gumbelSoftmax trick [231, 232]. Thus, each attention head selects one `HNSA`.

4.3.7 Learning to Forward

The interpretation of the NA's output $y \in [0,1]^{\hat{d}+1}$ depends on the loss function. In the case of $\mathcal{L}_{Ber}$, each output corresponds to the probability of success parameterizing a Bernoulli distribution. In the case of $\mathcal{L}_{Cat}$, each output corresponds to the probability of forwarding a flowlet f to any of the neighbors.

The output of the NN depends on the network state encoded in `HNSAs` , i.e., the output of the NSMod, the destination, and the switch that makes the forwarding decision. The NN learns to combine this information to react to changes in the state of the network and to learn how forwarding decisions differ for switches and destinations.

Calculating `Ports` requires the execution of the NSMod, the LocMod, and the FwdMod. The LocMod and the FwdMod each consist of one feed-forward NN with ReLU activation.

4.4 Node addressing schemes

To keep the management overhead of Mistill low, it would be beneficial to use the IP addresses of the nodes in the network as identifiers for hosts and switches instead of separate identifiers that introduce additional management overhead. Generally, IP addresses in communication networks can be expected to follow some form of pattern to keep routing tables small [233]. If IP addresses did not contain a useful pattern, e.g., are assigned randomly to every network device, routing tables would have to store forwarding decisions for every IP address. IP addresses and their patterns today are geared toward the available technology, e.g., towards longest prefix matching. It is unclear if the way nodes are addressed today is a good fit for learning distributed TE mechanisms.

Thus, this section investigates whether distributed TE mechanisms can be learned with IP addresses, whether latent space models, as used in natural language processing or graph learning, yield an addressing scheme that results in better performance, and whether the addressing scheme plays any role in the ability to learn forwarding behavior.

4.4.1 Learning binary embeddings of nodes in graphs

Representation learning for nodes in graphs is an area of active research [234]. This section investigates Bernoulli Embeddings [235] as node addresses. Bernoulli Embeddings map nodes into a binary space that encodes the neighborhood relationship. This section does not consider continuous latent space models such as `node2vec` [236] or additional node meta-data. The size of the binary representation of continuous embeddings is too large for addresses. Meta-data on the networking level, e.g., Medium Access Control (MAC) addresses, hardware information, etc., is either random or not correlated with the network structure.

A binary embedding E_i of node i is a d-dimensional binary vector $E_i \in \{0,1\}^d$. The binary vector is sampled from d independent Bernoulli distributions, parameterized by a matrix $\Theta \in \mathbb{R}^{|\mathcal{V}|\times d}$. The k$^{\text{th}}$ element in the embedding of node i is thus sampled according to: $E_i^k \sim \text{Bernoulli}(\Theta_{\text{i,k}})$. The embedding can be optimized by changing the parameterization of the independent Bernoulli distributions such that an appropriate loss function is minimized.

The method in [235] encodes the neighborhood relation into the embeddings by casting the problem as a link prediction problem: The probability of an edge e_{ij} from node i to node j is proportional to the hamming distance d_H between the embeddings E_i and E_j of those nodes:

$$p(e_{ij} \mid E_i, E_j) = \frac{\exp\left(-\alpha d_H(E_i, E_j)\right)}{\sum_{k\in\mathcal{V}} \exp\left(-\alpha d_H(E_i, E_k)\right)} \tag{4.30}$$

The parameter α is a scaling factor. The hamming distance d_H between two vectors is defined as: $d_H : \{0,1\}^d \times \{0,1\}^d \to N; x, y \to x^T(1-y) + (1-x)^T y$ [235].

The embeddings can then be trained by optimizing the log probability, i.e.:

$$\mathcal{L}(G, \Theta) = -\sum_{(i,j)\in\mathcal{E}} \mathbb{E}\left[\log p(e_{ij} \mid E_i, E_j)\right]_{\Theta}, \tag{4.31}$$

given the embeddings of incident nodes, assuming that edges are independent [235].

Misra et al. [235] use a continuous approximation to Eq. (4.31) to circumvent the expectation over a discrete-valued variable. In contrast, the Gumbel-Softmax trick [231, 232], a reparameterization trick for discrete random variables, is used in this chapter to optimize the objective in Eq. (4.31). In addition, Misra et al. [235] use Noise Contrastive Estimation (NCE) to avoid the summation over all nodes in the graph in the denominator of Eq. (4.30). This section does not use NCE since optimization is feasible for the size of communication networks. In addition,

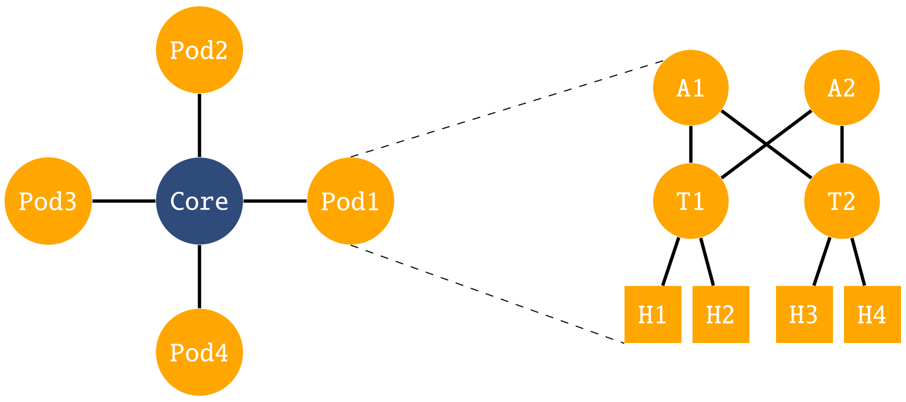


Figure 4.8 Graphs used to form the hierarchical embedding. The core and pods are represented as a star. Each star corresponds to one pod. Binary embeddings are learned separately for the star and *one* pod.

using NCE resulted in aliasing effects between nodes, making learning unique embeddings more difficult.

Addresses in communication networks must be unique, which translates to the embeddings of the nodes. The optimization objective in Eq. (4.31) is not designed for this constraint and might assign the same embedding to different nodes. In fact, for a fully connected graph, a single embedding for all nodes would be the best solution. Also, in other networks, nodes exist that are structurally equivalent and thus get the same embedding. For example, all hosts connected to one ToR switch in a Fat Tree topology would get the same embedding. Similarly, all core switches in a Fat Tree connected to the same aggregation switches are structurally equivalent and would get the same embedding. Eq. (4.32) thus adapts the objective to avoid duplicate embeddings. Specifically, the following energy function is used for the model of edge probabilities:

$$\mathcal{L}(G, \Theta) = -\alpha\sqrt{\left(d_H(E_i, E_j) - \beta\right)^2}, \tag{4.32}$$

where β is a scalar parameter, the intuition behind Eq. (4.32) is that a probability of one for an edge is obtained only if the embeddings differ. The parameter β controls the scaling of the difference. For example, $\beta = 1$ pushes the embeddings of adjacent nodes towards having a hamming distance of one.

4.4.2 Hierarchical Embedding

Fig. 4.1 shows that a Fat Tree architecture has a repetitive structure, i.e., identical pod structure. To exploit this pattern, this section designs a hierarchical embedding. Fig. 4.8 illustrates the principle for a $k = 4$ Fat Tree. The Fat Tree is abstracted into a star graph and a pod graph. The hub of the star represents the core switches. The star's satellites represent one pod each. Binary embeddings are then learned separately for the star and one pod structure. Core switches all get the embedding of the star graph's hub as a network address. Unique addresses can then be obtained by adding a separate host address.

For hosts, ToRs, and aggregation switches, the final embedding is obtained by concatenating the embedding for the pod node in the star graph with the embedding of the respective node's embedding from the pod graph. The embedding for node `T00` in Fig. 4.1 consists of the concatenation of E_{Pod1} and E_{T1}. The embeddings of pod nodes in the star graph act as a prefix to the nodes in the corresponding pods.

4.4.3 Hyperparameter Optimization

To obtain learning objectives, the CSP for ECMP is used to generate training data. The edge's attributes are static for the data generation, i.e., every edge is always available. Thus, only the LocMod is used to learn the forwarding behavior, and different NN architectures were evaluated: MHAs using the destination's embedding as a query and the neighbors' embeddings as keys and values, and feed-forward NNs. The results showed that a simple feed-forward NN with one hidden layer and concatenation of the destination's embedding and the current location's embedding is enough to learn the forwarding behavior in Fat Tree topologies reliably. Using Occham's Razor [27], i.e., preferring simpler models over complex ones, a feed-forward NN for the LocMod is used going forward.

The search space for the feed-forward NN is:

- batch size: $\{64, \ldots, 256\}$.
- hidden layer sizes: $\{8, \ldots, 100\}^3$, i.e., up to three layers.
- learning rate: $[0.001, 0.0001]$.
- samples for ASHA: 100.

The evaluation used the ADAM [237] optimizer and ReLU activation. The evaluation used the Asynchronous Successive Halving Algorithm (ASHA) [238] implemented in the Ray Tune [239] library for hyperparameter optimization and to select the best configuration for the evaluation in Sec. 4.4.4.

4.4.4 Results

The suitability of addressing schemes is evaluated by learning static shortest paths. An addressing scheme is suitable if the forwarding rules can be learned reliably, i.e., if the trained NN obtains a low loss value. Tbl. 4.1 lists the results obtained for this learning task, using addresses of up to 24 bits. For the hierarchical embedding, 12 bits are used for the embeddings of the star and pod graphs, respectively. IP addresses follow the scheme in [183]. The first octet is removed since it is constant and thus does not contribute any information.

Opt	Rnd	L12	L16	L20	L24	HR	IP
0.3707	0.6210	0.5397	0.4835	0.5002	0.6541	0.4185	0.3921

Table 4.1 Training results for learning shortest paths to evaluate embeddings. Optimal is the average loss of the ground truth. Opt. corresponds to the average loss over the ground truth, i.e., perfect predictions. Rnd. corresponds to random binary embeddings. L12-L24 corresponds to learned binary embeddings, HR to the hierarchical embedding scheme, and IP to IP-address-based embeddings.

The random embedding (Rnd in Tbl. 4.1) results in the highest, i.e., worst, loss value. The addresses do not contain any form of a pattern, after all. The learned embeddings (L12-L24 in Tbl. 4.1) result in lower loss values. The dimensionality of the embedding does not affect performance. No consistent increasing or decreasing trend is observable. The hierarchical embedding (HR in Tbl. 4.1) achieves the second-best loss, surpassed only by the IP address-based embedding results. Good performance for the IP address-based embeddings is explainable since the forwarding behavior can be perfectly computed from the IP address alone in the case of the Fat Tree. The results for LocMod architectures other than feed-forward NNs show similar behaviors.

In summary, IP addresses are well suited to learning forwarding decisions on well-structured topologies such as Fat Trees. This has the advantage that learned distributed TE mechanisms are backward compatible, can be incrementally deployed or run side-by-side with traditional protocols. Therefore, Mistill uses the last 24 bits of the IP addresses to represent nodes.

The results also show that obtaining ML-friendly IP addresses in general graphs could be an interesting avenue of future work. This section's results show that learning embeddings, at least with the presented scheme, did not produce good predictive performance. Only introducing a bias in the structure through the hierarchical embedding resulted in a competitive performance to the IP addresses. However, general graphs might not have such a hierarchical and repetitive structure. Deriving other schemes to learn addresses that enable the learning of forwarding decisions might thus be necessary.

4.5 Evaluating learned policies

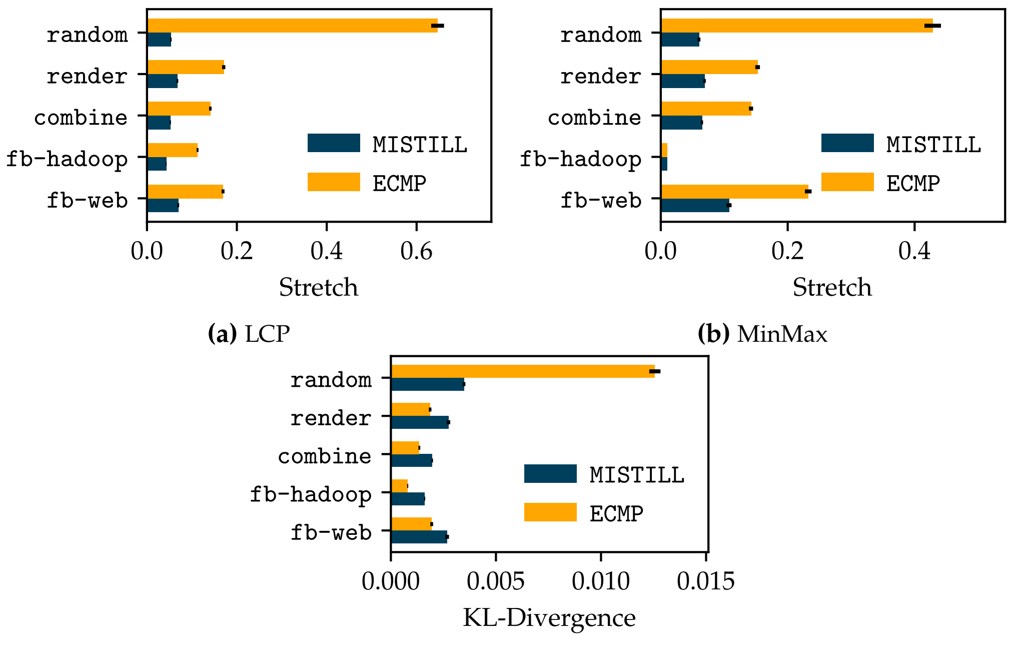
This section evaluates Mistill's performance for the TE policies `WCMP`, `LCP`, and `MinMax`. The policies can be interpreted to protect the DCN from traffic whose destination is unreachable. If no route to a host exists, i.e., if the corresponding CSP or COPTP is infeasible, then traffic is dropped. The challenge lies in accounting for link failures in the up-link from hosts to switches, which existing TE schemes cannot handle. Dropping traffic can be important for applications that send larger amounts of data with UDP. Existing TE schemes would forward this traffic to the last hop of the network and thus waste bandwidth [182, 184]. Fur-

	HNSAMod			LocMod	NSMod					FwdMod
	FCL	#Distr.	Arity	FCL	#Heads	W_q	W_k	W_v	FCL	FCLs
MinMax	108	64	2	90	14	48	41	17	109	104, 84
LCP	63	64	2	90	14	48	48	50	125	102, 105
WCMP	48	64	2	90	14	62	62	27	110	76, 71

Table 4.2 Hyper parameters for the best models.

ther, the learned TE mechanisms should work in the presence of arbitrary node and link failures. This section evaluates Mistill's performances to ECMP's and the actual forwarding decision. This section largely corresponds to MaLANe's *Train Model* activity and touches on the *Generate Data*, *Investigate Data*, and *Prepare Training* activity.

4.5.1 Data Generation and NN training



(a) LCP **(b)** MinMax **(c)** WCMP

Figure 4.9 Bar plots for TE policies and traffic patterns.

Data for a $k = 8$ Fat Tree having 128 hosts, 80 switches, and 768 directed links is generated. To make the protocols robust to node and link failures, between zero and ten edges and zero and five nodes are removed uniformly at random from the topology. For the remaining edges, weights are sampled according to five patterns: `random`, `combine`, `render`, `fb-hadoop`, and `fb-web`. In `random`, the cost of each edge is sampled from a uniform distribution. The other patterns are created from traffic matrices of data center applications. `fb-hadoop` is based on a Hadoop cluster, `fb-web` is based on production traffic from a Facebook data center [172]. `combine` results from a web-search, and `render` renders the search results for users [240]. Note that `combine, render, fb-hadoop` and `fb-web` result in utilization patterns that strongly differ from `random`.

For training, only edge costs sampled according to `random` are used. To obtain forwarding decisions, source and destination pairs are sampled, and solutions for each policy are obtained on the generated graph for the sampled pairs. Target variables are then constructed from the resulting solutions. The training and validation sets consist of 200 000 and 50 000 source and destination pairs.

The final results are obtained from a packet-level simulation. For each packet, the metric of the optimal path according to the respective TE policy is compared with the metric of the path from the NN and `ECMP` using the stretch of the path. The stretch is defined as $\frac{|c - c_{OPT}|}{c_{OPT}}$, where c is the metric of the evaluated NN, and c_{OPT} the optimal metric. The stretch is the difference between the chosen and the optimal path relative to the optimal path.

4.5.2 Hyperparamter Optimization

Extensive parameter sweeps are performed with `ASHA` to learn the forwarding behavior with the network state. For the LocMod, the best-performing architecture for the shortest-path task in Sec. 4.4 is used, i.e., a feed-forward NN with one hidden layer having 90 neurons. For the other modules, `ASHA` explores the following parameter space:

- batch size: $\{64, \ldots, 128\}$,
- final Fully Connected Layers (FCLs) in the FwdMod: $\{50, \ldots, 128\}^{\{2,3\}}$,
- FCL for encoding links: $\{32, \ldots, 150\}$,
- number of blocks: $\{1, \ldots, 65\}$,
- arity: $\{2, \ldots, 16\}$,
- tempererature of gumbelSoftmax: 0.6,
- number of heads for `MHA` in NSMod: $\{9, \ldots, 14\}$,
- dimension of FCL for `MHA` module in NSMod: $\{70, \ldots, 129\}$,
- hidden dimension of `MHA` heads: $\{20, \ldots, 64\}$,
- output dimension of `MHA` heads: $\{16, \ldots, 4\}$,
- learning rate: $[10^{-4.3}, 10^{-3.5}]$.

To obtain the final models, the arity of the categorical distributions in the HNSAMod is set to 2 and the number of blocks to 64. `ASHA` then trains 100 models, resulting in the parameters in Tbl. 4.2. To evaluate the impact of the number of blocks, the parameters in Tbl. 4.2 are kept fixed, and the number of blocks is varied. For all policies, the HNSAMod consists of one fully connected layer with ReLU activation and a linear transformation that generates the logits for the gumbelSoftmax function. The LocMod consists of one fully connected layer with ReLU activation. The NSMod consists of one `MHA` module and the FwdMod of two fully connected layers. Training a single model takes ~5 hours on an NVIDIA Tesla V100 GPU. Training 100 models in parallel with `ASHA` on three NVIDIA Tesla V100 GPUs takes 60 hours.

4.5.3 Routing Performance

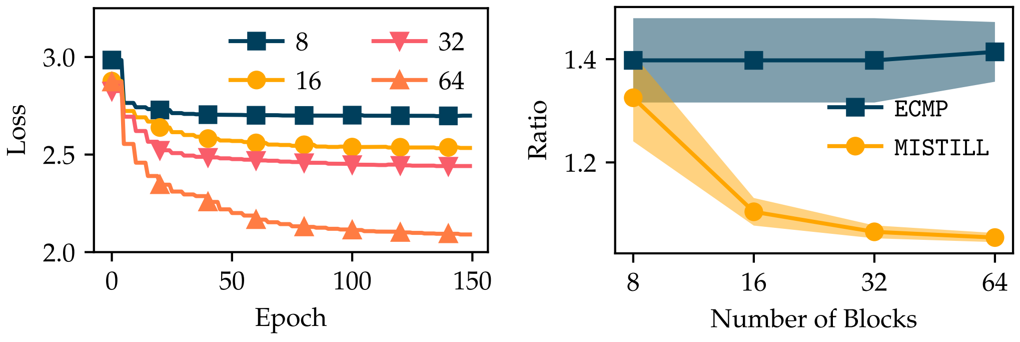
After training on edge costs generated with `random`, the stretch of the trained NNs for `LCP`, `MinMax`, and `WCMP` is evaluated for each of the five traffic patterns in 100 simulations.

LCP. Fig. 4.9a shows bar plots for the `LCP` policy and compares Mistill to `ECMP`. The Y-Axis shows the average stretch of the objective. Error bars correspond to the 99 % confidence interval of the mean. Fig. 4.9a shows that Mistill is better for all traffic patterns, with paths whose weight is close to the optimal weight. Further, Fig. 4.9a shows that the error of Mistill is similar across all traffic patterns. The NN thus generalizes to distributions of the inputs that it has never seen during training.

MinMax. Fig. 4.9b shows a similar plot as in Fig. 4.9a for the `MinMax` policy. Fig. 4.9b shows that Mistill is always better than `ECMP`, except for `fb-hadoop`. For `fb-hadoop`, Mistill's and `ECMP`'s stretch is almost zero. This is due to the specific pattern in `fb-hadoop`, in which the uplinks of hosts are the most loaded links, and the TE policy has thus almost no impact on the objective. As for `LCP`, the NN generalizes to unseen input distributions.

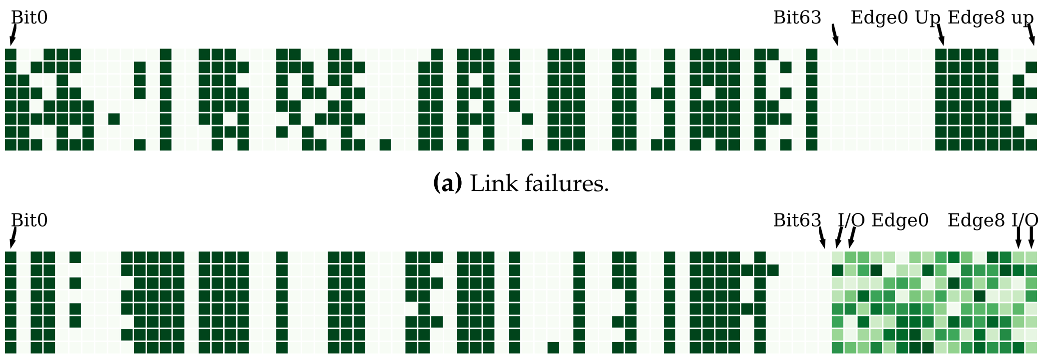
WCMP. The Kullback-Leibler Divergence (KLD) is used to evaluate `WCMP`. Fig. 4.9c shows the average KLD for Mistill and `ECMP` for the traffic patterns. Error bars correspond to the 99 % confidence interval of the mean. Fig. 4.9c shows that Mistill is better than `ECMP` for the `random` traffic pattern. `ECMP` has slightly smaller average KLDs for the other patterns. This is because available bandwidth is almost equally distributed over paths. `WCMP` essentially becomes `ECMP` up to a small error. Note how the average KLDs of Mistill decrease for the other traffic patterns. This indicates that the NN also generalizes to unseen input distributions.

Summary The experiments show that Mistill can learn the forwarding behavior of three popular TE policies from exemplary samples. Further, the results show that the NN generalizes to previously unseen input distributions when trained with random uniform edge weights.



(a) Training Progress. **(b)** Performance.

Figure 4.10 Training loss for a varying number of blocks in the HNSAMod for the calculation of HNSAs.



(a) Link failures.

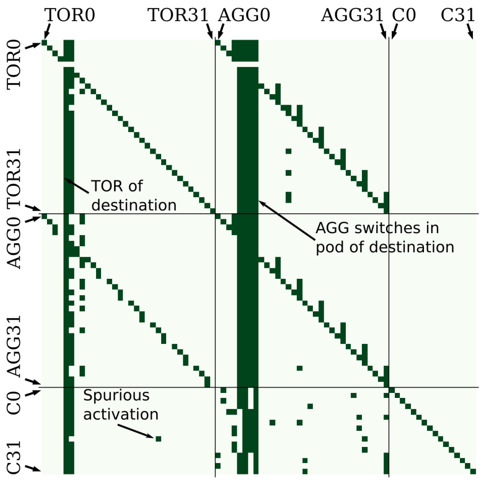
(b) Edge Weights.

Figure 4.11 Bit pattern in HNSA messages for edge availability and edge weights.

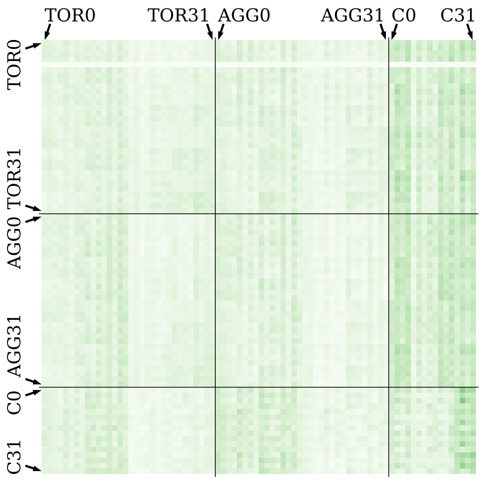
4.5.4 HNSA Size

Fig. 4.10a illustrates the impact of the number of categorical distributions on the MinMax policy by varying them in $\{8, 16, 32, 64\}$. Fig. 4.10a shows that more distributions result in a smaller loss and that the decrease in loss is strong during the initial stages of training and converges towards the end.

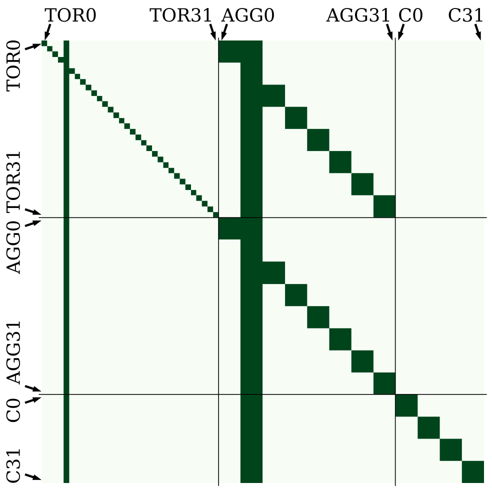
Fig. 4.10b compares the ratio between ECMP and the true solution to the ratio of Mistill. The solid line corresponds to the mean, and the shaded area to the mean's 99 % confidence interval. Fig. 4.10b shows that 8 distributions result in a NN that performs better than ECMP. The NN with 8 distributions, i.e., 8 bits, mostly account for link failures and little for edge weights. The performance of Mistill improves with more distributions and approaches the ground truth, and the confidence interval shrinks. The improvement between 8 and 16 distributions is larger than between 16 and 64. With 64 distributions, Mistill selects paths with similar metrics to the ground truth. The resulting update messages in the network have thus a payload of 8 Byte, which corresponds to a reduction by a factor of 24 compared to the original input.



(a) gumbelSoftmax



(b) sparsemax



(c) Edge Cover

Figure 4.12 Attention weights. Each row contains the weights for one switch over all other switches.

4.5.5 Visualizations

This section visualizes the learned HNSA messages and the attention weights. Fig. 4.5.5 compares the attention weights of the gumbelSoftmax activation with the weights of a model trained with the sparsemax activation function [241] for MinMax.

HNSA content Fig. 4.11 shows HNSAs of Mistill for the MinMax policy. Fig. 4.11a illustrates how link failures reflect in the HNSAs. Fig. 4.11b illustrates how edge weights reflect in the HNSAs. The left part of Fig. 4.11a and Fig. 4.11b show the HNSAs. The right part shows the availability of edges in Fig. 4.11a and the edge weights in Fig. 4.11b. White corresponds to a value of zero and green to a value of one.

Fig. 4.11a shows that many of the bits encode the link availability, i.e., 56 out of 64 bits can be associated with link availability. Some bits even directly correspond to the availability of edges. Bit59 and Bit26 correspond to Edge8, and Bit4 to Edge7.

Consequently, Fig. 4.11b shows that most bits are constant for different edge weights. The NN uses eight out of the 64 bits to encode edge weights. The relation between bits and edge weights is more complex than link availability, and no clear correspondence is observable.

Attention Scores Fig. 4.12 illustrates the attention scores of all switches in the network for H16, i.e., the first host in the second pod. To evaluate the effect of the gumbelSoftmax activation, an additional NN with the sparsemax activation function [241] is trained. Further, Fig 4.12c shows the attention scores corresponding to an edge cover of all edges belonging to the shortest paths between a switch and the destination. Fig. 4.12a shows the attention scores of the gumbelSoftmax activation and Fig. 4.12b of the sparsemax activation. The attention scores of individual attention heads are summed up and clipped to a maximum value of one.

Fig. 4.12a shows that the attention scores of the gumbelSoftmax resemble the edge cover in Fig. 4.12c. In contrast, the attention scores of the sparsemax cover all nodes. Fig. 4.12a shows that the NN relies on the state of the ToR switch- and the aggregation switches in the destination pod. Further, all switches rely on their state. Some activations in Fig. 4.12a are not intuitive. For example, it is unclear why core switch C10 relies on the state of a ToR switch in a different pod than the destination. This could be a training artifact, i.e., a spurious activation, and potentially avoided through regularization of the attention scores.

Fig. 4.12a and 4.12b show that the sparsemax activation results in a dense pattern that results in a large signaling overhead of the learned TE mechanism. The resulting mechanism would exchange one message for each non-zero entry. The gumbelSoftmax activation has a clear advantage and requires the exchange

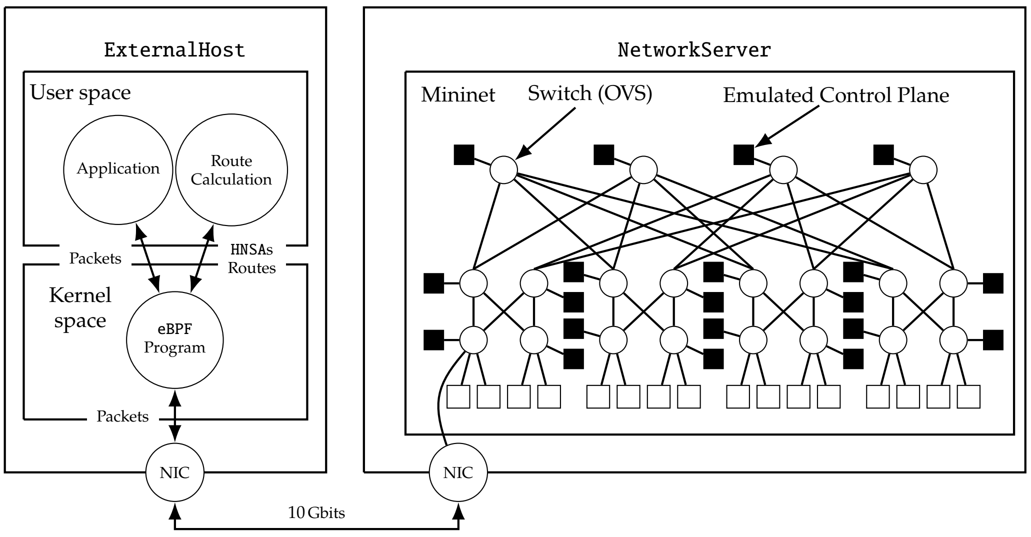


Figure 4.13 The PoC consists of two servers connected back-to-back with 10 Gbit/s. The right server emulates a Fat Tree topology with Mininet. The left server acts as a host connected to the first ToR switch in the first pod.

of at most 10 messages, which is in the order of the edge cover, which requires at most 9 messages.

In summary, most bits in the `HNSA`s encode the edge availability. The attention scores resemble an edge cover of edges from the shortest paths to a destination. The detected pattern is sparse, which the PoC exploits to improve performance. The behavior for `WCMP` and `LCP` is analogous.

4.6 Proof-of-Concept Implementation

This section describes and evaluates an end host-based PoC of Mistill in Mininet, focusing on systems engineering-related performance indicators. The PoC is *not* intended for accurate network-wide performance evaluations, e.g., measuring flow completion times, packet drop rates, congestion, etc. Measuring these values in the PoC would have little utility since the results are not transferable to an actual network and depend on use-case-dependent parameters such as the traffic pattern. Instead, the primary purpose of the PoC is to quantify the data plane overhead, the NN inference speed, the impact of the NN implementation on the inference speed, and the time Mistill needs to react to changes in a switch's local state. This section reflects the MaLANe's *Integrate Model* and *Deploy ML-enabled System* activities.

4.6.1 Overview

The PoC builds on Mininet and uses the extended Berkeley Packet Filter (eBPF), Multiprotocol Label Switching (MPLS)-based Segment Routing (SR), and IP Multicasting (IPMC) to implement Mistill. Fig. 4.13 illustrates the PoC architecture.

The PoC consists of two servers connected back-to-back. The `NetworkServer` emulates a Fat Tree topology with Mininet, the `ExternalHost` is connected as end host to the first ToR in the first pod of the emulated Fat Tree. The PoC uses MPLS to implement SR, using path segments with a length of at most two hops[9]. `ExternalHost` serves as an SR-ingress node and uses the eBPF to encapsulate outgoing packets into an SR tunnel. It uses the NN to determine the tunnels from the switches' `HNSA`s. The PoC connects one Mininet host to each switch (black squares in Fig. 4.13). Each such host emulates a switch-local control plane that monitors its switch's ports, calculates `HNSA`s, and sends them out as an IPMC packet. The PoC installs the necessary rules for path segments and IPMC using `OpenFlow` statically on the switches.

4.6.2 Design Decisions

This section justifies the major design decisions in the PoC.

Why Segment routing? SR has two advantages. 1) SR minimizes the requirements Mistill has towards the network. 2) The route computation, involving most parts of Mistill's NN, moves into the end hosts.

The requirements towards the DCN reduce since static rules suffice to forward traffic. Further, support of SR is ubiquitous and possible with today's network devices. Administrators can implement SR with e.g., MPLS, and IPv6 [242]. This makes Mistill amenable for deployment in legacy networks.

The CPUs in end hosts have accelerators for ML inference. For example, the AVX set has been available since 2011 [226]. Further, ML acceleration is incorporated into Network Interface Cards (NICs) [243, 244]. In contrast, switches are not designed for ML inference [56–58]. Even recently proposed architectures designed for in-network ML execute only small [58] or binarized NNs [57]. The PoC thus aims to minimize the ML inference on switches as much as possible.

Lastly, predicting each switch's forwarding decision on the hosts allows the learning of TE policies for which the Markov Condition as described in Sec. 4.3.2 does not hold. Hosts can easily track and provide the decision or state history, e.g., the encountered maximum link capacity. If switches make forwarding decisions, this information would have to be transmitted along with the packet.

Why eBPF? The eBPF is a native part of the Linux Kernel [245], making Mistill easy to maintain, i.e., Mistill does not depend on hard-to-maintain changes to the Operating System[10]. Further, eBPF supports NIC offloading [244], enabling the execution of Mistill directly on the NIC.

[9]The OpenVSwitch (OVS) supports an MPLS label stack size of at most three. Paths between hosts in separate pods have a length of five. Thus, the PoC has to encode two hops into one label.
[10]eBPF is even becoming available under Windows [246].

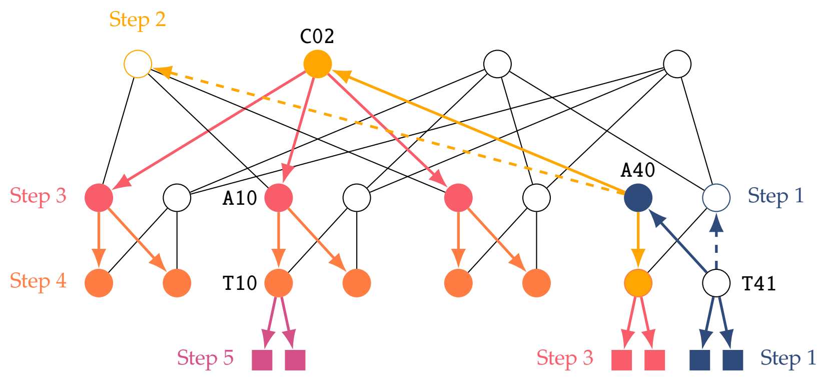


Figure 4.14 Probing scheme for all node types.

Why Multicast? The PoC uses IPMC of HNSAs for three reasons: IPMC reduces overhead compared to broadcasting, IPMC allows the dynamic subscription of end hosts to a switch's HNSAs, and IPMC is easy to deploy in Fat Tree topologies.

IPMC uses a tree where the sender is the root, and the receivers are the leaves. Intermediate nodes in the tree duplicate messages in the downstream direction only. In the case of the PoC, switches correspond to sources and end hosts to receivers. Thus, switches duplicate packets at most over half their neighbors, thus reducing the absolute number of update messages compared to broadcasting.

IPMC allows receivers to join IPMC groups dynamically. Thus, end hosts can subscribe to those switches' updates that they need according to the SI output, allowing the automatic adaption of update message exchange to traffic patterns.

The Fat Tree topology is well suited for IPMC. A Fat Tree topology is a multi-rooted tree [183]. Together with the Fat Tree IP addressing [183], a static IPMC scheme requires only a few rules in an OpenFlow switch.

4.6.3 HNSA computation

Switches in the network compute HNSAs with the HNSAMod from their local state. Each switch monitors its ports and collects the statistics at a rate λ_{LS}. In the PoC, switches monitor the ports' utilization and availability.

The PoC emulates the local control plane through a separate host connected to each switch in the network. Each emulated control plane runs a C++ program that encodes the local state, computes HNSAs with a copy of the HNSAMod, and sends the HNSAs into the network.

4.6.4 HNSA dissemination

Fig. 4.14 illustrates exemplary forwarding of IPMC messages in the network. Fig. 4.14 divides the forwarding into five steps. Solid arrows indicate the sending of a packet. Dashed arrows indicate alternative paths, and T41 sends an update message.

In step one, `T41` duplicates the message downstream, i.e., sends a copy of the message to each host, and *one* copy to a random upstream node, `A40` in Fig. 4.14. In step two, `A40` duplicates the message to all connected ToR switches, except the ToR switch the message came from. Further, `A40` randomly sends the message to *one* core switch, `C02` in Fig. 4.14. In step three, `C02` duplicates the message to all connected aggregation switches, except the aggregation switch it received the message from. In addition, the ToR switches that received the message in step two duplicate the message downstream to their hosts. Thus, after the third step, all hosts in the pod the message originated in have received the message. In step four, all aggregation switches that received the message in step three duplicate the message to the connected ToR switches. In step five, the ToR switches duplicate the message to the connected hosts.

Messages from core, aggregation, and other ToRs are treated analogous, with the only exception that aggregation and core switches duplicate their message to *all* downstream nodes. The described dissemination scheme is resilient to link failures and has little overhead.

The update distribution can mitigate link failures because multiple upstream paths are available. Randomly sampling upstream nodes ensures that update messages arrive, i.e., only a fraction of the updates is affected.

The overhead wrt. link capacity is:

$$\frac{8s\frac{5}{4}k^2\lambda_{LS}^{-1}}{l}, \tag{4.33}$$

where l is the link speed in Bit/s, and s the packet size in Bytes. In the PoC with $k = 8$, $l = 1\,\text{GBit/s} = 1e^9\,\text{Bit/s}$, $s = 62\,\text{Byte}$[11], $\lambda_{LS} = 1000\,\text{Hz}$, and all-to-all traffic, the overhead is 3.97 %. The overhead is larger than that of `HULA`, which has an overhead of 0.16 % in this setting [175]. `HULA` benefits from the custom P4 implementation that converges information exchange with route update calculations, resulting in reduced information exchange. The design decision for Mistill to keep the potential space of learnable TE policies unrestricted comes at the price of higher overhead.

4.6.5 Host Implementation

The host implementation has two programs: one runs in the kernel-, and the other in the user space. Fig. 4.15 illustrates the programs. The kernel- and user space communicate via three eBPF maps. The `NetworkState` map contains the received `HNSA`s. The `Routes` map contains updated routes to destinations. The `ActiveDsts` map contains the currently active destinations, i.e., destinations the host sends traffic to.

[11] `HNSA`s are UDP packets with 8 Bytes payload: 8 Bytes preamble, 14 Bytes ethernet header, 20 Bytes IP header, 8 Bytes UDP header, 8 Bytes `HNSA`, and 4 Bytes frame check sequence. Note that UDP is not mandatory.

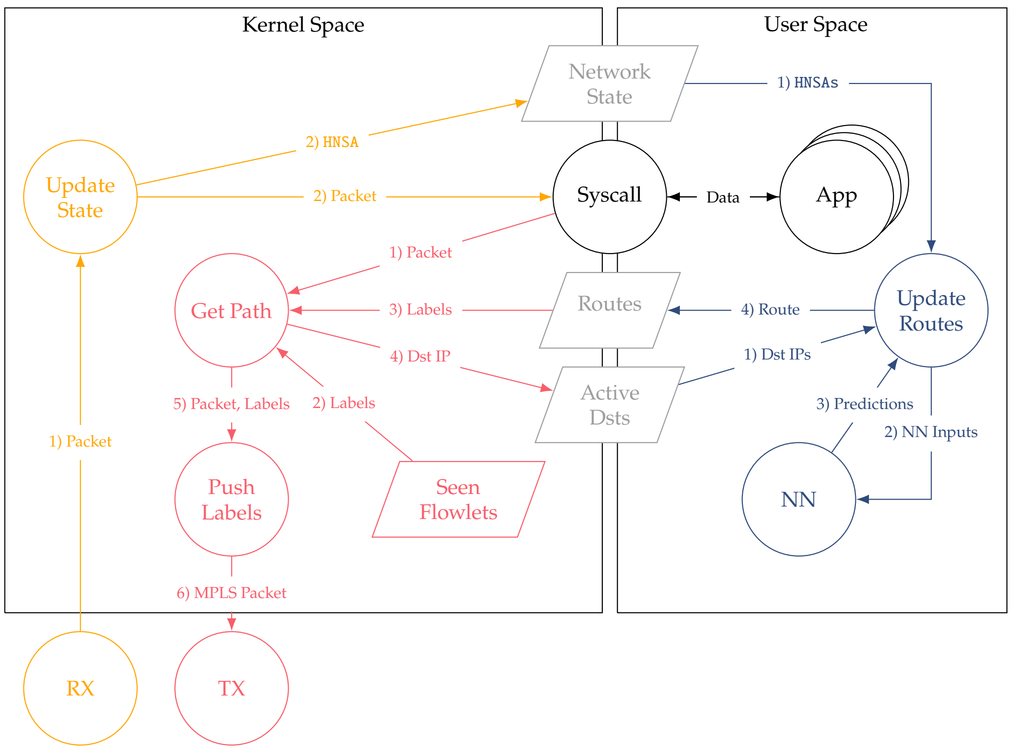


Figure 4.15 Diagram of the host implementation. Yellow corresponds to ingress, red to egress, and blue to route updates.

The user space program is written in C++ and updates the routes for the active destinations, i.e., the hosts this host communicates with. For this, the user space program reads the HNSAs from the NetworkState map, the active destinations from the ActiveDsts map, and constructs the inputs for the NN. Then, the user space program executes the NN and constructs routes from the NN's outputs, and writes the new routes to the Routes map.

The kernel space implementation consists of two eBPF programs written in C. One program handles traffic in the egress (TX) and the other in the ingress (RX). Both programs operate in the traffic control (tc) layer. In the ingress, the program receives a packet from the NIC and checks if the packet is an IPMC packet carrying a HNSA. If yes, the program extracts the HNSA from the packet's payload, updates the NetworkState map, and drops the packet. If no, the program returns the packet to the kernel.

In the egress, the program receives a packet from the kernel. The program checks if the packet belongs to an active flowlet, i.e., to a five-tuple the program has already seen within a configurable amount of time. If the packet belongs to an active flowlet, the program retrieves the route from the SeenFlowlets map and updates the time stamp of the last match. If the packet does not belong to an active flowlet, i.e., the entry in the SeenFlowlets timed out, or the map does not contain the five-tuple, then the program retrieves a route from the Routes map and updates the SeenFlowlets map accordingly. As a result, the program will forward packets of a flowlet over the same route, preserving packet order at the receiver. After retrieving a route, the program adds the corresponding MPLS labels to the packet. Finally, the program returns the packet to the kernel.

The kernel- and the user space programs operate independently from each other. The programs communicate state and updates through the shared eBPF maps. This approach keeps the overhead in the data plane low, as Sec. 4.6.7 shows.

4.6.6 NN Implementation

The NN implementation for inference is crucial for the overall performance of NN-based routing protocols. Existing TE protocols can react within milliseconds to changes in the network state [175, 180, 182]. The NN inference time thus determines if NN-based routing protocols can be competitive. To obtain high performance and assess the impact of the NN implementation on the overall performance, the forward pass of the NN is implemented in `C++`. Further, following the guidelines of deploying NNs for performance-critical applications, TorchScript and the `libtorch` library is used to access the saved PyTorch models [247].

The custom `C++` implementation enables the memory layout optimization for better cache utilization and the reduction of computations by exploiting NN-specific structures, e.g., the sparsity in attention scores in the NSMod. The custom implementation uses the Advanced Vector Instructions 512 (AVX512) instruction set on the `ExternalHost` and no acceleration on the `NetworkServer`. The NN is single-threaded on both machines. In contrast, `libtorch` integrates with optimized libraries for linear algebra that can exploit multi-core systems and hardware accelerators such as GPUs. Further, PyTorch can optimize the NN through layer fusion and optimizations in the computational graph [247]. All implementations take the maximum over the output layer to determine the neighbor to which a switch should forward a flowlet.

4.6.7 Evaluation

The NN inference time is measured on the external host, as well as the execution time of the user space program, the impact of the NN implementation on inference speed, the data plane overhead in ingress and egress, and how fast the PoC reacts to changes in the network state.

Mininet emulates a $k = 8$ Fat Tree with 80 switches, 128 hosts, and 768 directed links with 1 Gbps on the `NetworkServer`. Each switch executes the NSMod on CPUs[12]. The `ExternalHost` runs the eBPF program and is connected to `T00`. The `NetworkServer` has two Intel(R) Xeon(R) E5-2650 v4 CPUs. The `ExternalHost` has one Intel(R) Core(TM) i7-7820X CPUs and two NVIDIA Titan XP GPUs, of which one GPU is used. The NN trained for the `MinMax` policy is evaluated. The timings are measured in wall-clock time with the `bpf_ktime_get_ns()` in the

[12]The inference speed on the GPU is not evaluated since the NSMod is small and cannot benefit from batching. Thus, the communication overhead between the main- and GPU memory can even lead to longer inference times, as the results in this chapter suggest.

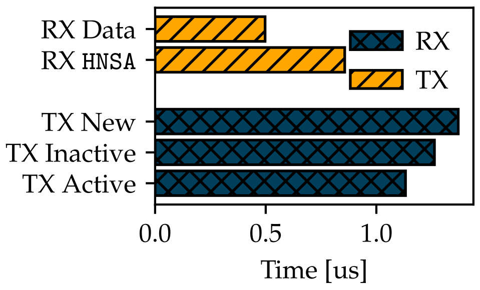


Figure 4.16 Overhead of the Kernel space program.

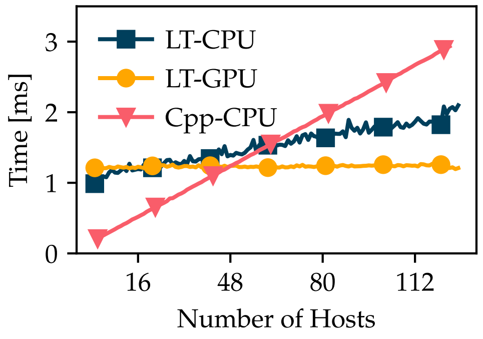


Figure 4.17 NN inference time for $\{1, ..., 128\}$ hosts.

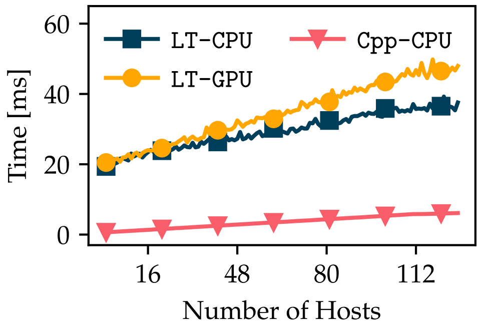


Figure 4.18 User space execution time for $\{1, ..., 128\}$ hosts.

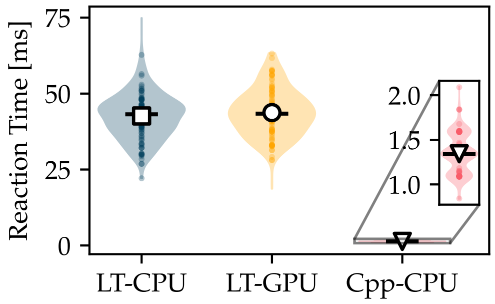


Figure 4.19 Reaction speed after a link failure.

eBPF programs, and the `clock_gettime()` function with the `CLOCK_MONOTONIC` clock in the user space program. Wall-clock time is measured since it simplifies the comparison of NN inference speed on heterogeneous hardware, and wall-clock time is what counts in practice.

Data plane overhead. The eBPF programs' overhead in the data plane on RX and TX for the custom NN implementation (`Cpp-CPU`) for the individual execution branches explained in Sec. 4.6.5 is measured. Fig. 4.16 shows the average processing time as a bar plot. The behavior and values for the `libtorch` based implementations on CPUs (`LT-CPU`) and GPU (`LT-GPU`) are comparable. Fig. 4.16 shows that the overheads vary between 0.5 µs and 1.5 µs. The overheads are negligible compared to overall latencies in the Linux networking stack [248, 249]. In RX, processing data packets takes 0.45 µs, and processing of `HNSA` packets 0.85 µs on average. In TX, the processing of new flowlets takes the longest with 1.37 µs, followed by the processing of inactive flowlets with 1.26 µs, and of active flowlets with 1.13 µs on average. This behavior is expected and reflects the different computational efforts for each case.

NN inference time. Fig. 4.17 shows the average inference times for `Cpp-CPU`, `LT-CPU`, and `LT-GPU` for 1 to 128 destinations. Fig. 4.17 shows that the inference time increases linearly for `Cpp-CPU` from 0.21 ms to 2.93 ms and for `LT-CPU` from

0.99 ms to 2.10 ms. The inference time for LT-GPU remains constant at ~1.21 ms. The results show that the Cpp-CPU implementation is faster for up to 48 destinations. After that, the libtorch-based implementations benefit from utilizing multiple cores.

Route update and HNSA calculation latency. Fig. 4.18 shows a line plot for the route update latency t_{ru}, i.e., the average execution time for the user space program. Fig. 4.18 shows that Cpp-CPU is fastest with 0.65 ms for one destination and 6.13 ms for 128 destinations. LT-CPU takes from 19.38 ms to 37.57 ms, and LT-GPU from 20.50 ms to 47.96 ms.

The high t_{ru} for LT-CPU and LT-GPU are unexpected. The long duration results from interacting with libtorch's tensor classes. Updating the input tensors alone takes ~15 ms. The reason presumably lies in the sequential setting and retrieving of individual tensor elements. The input data is not readily available as a larger memory block that libtorch could wrap inside a Tensor. Similarly, the PoC must process outputs individually to convert them into forwarding decisions. The higher t_{ru} values for LT-GPU compared to LT-CPU are unsurprising. Inference on the GPU entails a communication overhead for copying data between main memory and GPU, which explains the higher t_{ru} values compared to LT-CPU.

Similarly, the HNSA calculation latency t_{HNSA}, i.e., the time it takes to calculate and transmit one HNSA, for Cpp-CPU is lower than for LT-CPU on the NetworkServer. On average, the computation of a HNSA's content takes 5.10 ms for LT-CPU and 0.03 ms for CPP-CPU.

This evaluation shows that small NN inference times are necessary but insufficient to achieve high-performance NN-based TE mechanisms. Auxiliary data processing can easily become a bottleneck and must be accounted for.

Reaction time. The reaction time is evaluated by measuring how fast MISTILL reacts to a link failure between ToR-00 and Agg-00, denoted by t_{rt}. Analytically, the expected reaction time $\mathbb{E}[t_{rs}]$ is modeled as:

$$\mathbb{E}[t_{rs}] \doteq \frac{1}{2\lambda_{LS}} + \mathbb{E}[t_{HNSA}] + \mathbb{E}[t_{Net}] + 1.5\mathbb{E}[t_{ru}] + \frac{1}{2\lambda_{pps}}, \tag{4.34}$$

where t_{Net} corresponds to the transmission and propagation delays and the time to update the NetworkState map.

To measure t_{rt}, the PoC sends packets at a frequency of λ_{pps} from the server ExternalHost to the emulated host H16. The timeout gap for flowlets is set to zero, i.e., the TX program retrieves the route for each packet from the Routes map. The utilization of link (T00, A00) is artificially set to 0.01, of (T00, A01) to 0.3, and of (T00, A02) and (T00, A03) to 0.9. As a result, the NN will send traffic via A00 if the link (T00, A00) is available. If the link fails, the NN will send traffic via A01. T00 samples its local state at a frequency of λ_{LS}. The PoC measures the time between changing the state of the link (`T00`, `A00`) and the first packet taking the correspondingly updated route. For `LT-CPU` and `LT-GPU` the PoC uses $\lambda_{pps} = 1\,000\,\mathrm{Hz}$ and $\lambda_{LS} = 100\,\mathrm{Hz}$. For `CPP-CPU` the PoC uses $\lambda_{pps} = 8\,000\,\mathrm{Hz}$ and $\lambda_{LS} = 4\,000\,\mathrm{Hz}$. The frequencies for the `libtoch`-based NNs are smaller to account for the longer NN execution times.

Fig. 4.19 shows violin plots for `LT-CPU`, `LT-GPU`, and `Cpp-CPU`. The average reaction times are 42.27,ms, 43.75 ms, and 1.34 ms, respectively. The empirical values are close to the expected values according to Eq. (4.34). t_{rt} is dominated by the route update latency and the `HNSA` calculation for `LT-CPU` and `LT-GPU`. In contrast, $\mathbb{E}[t_{Net}]$ accounts for 24.99 % of `Cpp-CPU`'s t_{rt}. The dominance of $\mathbb{E}[t_{Net}]$ is not unexpected, considering that the network on the `NetworkServer` is emulated, and the switches are implemented in software. The results for `Cpp-CPU` are competitive with state-of-the-art in-network TE mechanisms. `HULA` and `CONTRA` report reaction times of 0.8 ms with hardware switches.

4.7 Discussion

This chapter provides the first quantification of NN-based TE mechanisms for DCNs. Thus, this chapter focuses on a full-stack implementation of NN-based TE mechanisms from NN design to the actual realization on hardware. NNs can learn distributed TE mechanisms for Clos Topologies from exemplary forwarding decisions. The resulting NN-based TE mechanisms can be implemented with negligible data plane overhead based on SR in legacy networks and react to network state changes within milliseconds. The work in this chapter thus establishes a strong foundation for future work on NN-based TE mechanisms. Specifically, this chapter shows for the first time that such mechanisms can meet stringent performance requirements. This chapter leaves important aspects for future work: The implementation of the control plane on physical switches, the scaling of the NN to larger topologies, the evaluation of NN optimization techniques, and the investigation of NN offloading.

Port statistics are readily available in the PoC, and monitoring at a high frequency is possible. Port statistics might be available at lower sampling intervals on real switches, especially on legacy equipment, which would reduce t_{rt}. Other NNs for the HNSAMod, e.g., Recurrent Neural Networks [250], could help to overcome this limitation by predicting values between samples.

Based on the learned attention weights in Fig. 4.12, it is expected that the NA can handle larger topologies. The number of attention heads in the NSMod is expected to grow linearly with the number k, i.e., the number of pods. Thus, there is reason to believe that the proposed NA has the potential to operate networks with thousands of hosts.

The PoC uses single-precision floating point numbers. However, NNs are often converted to lower precision or integer quantized for production. Lower precision and quantization decrease the memory consumption and NN execution times [251, 252]. Further, Deep Learning Compilers can optimize a given NN for different hardware platforms [253]. Thus, the results of this chapter on the inference speed can likely be further improved.

Network Equipment like Data Processing Unit (DPU) could allow the offloading of NN inference to NICs, allowing an integration of the PoC's user-space program into kernel space. This could further improve the performance, specifically given the accelerators available on DPUs.

4.8 Conclusion

This chapter presents Mistill that learns distributed TE mechanisms from a TE policy's exemplary forwarding decisions with ML. Mistill learns a NN that represents the processing and exchange of local information with other network elements and the computation of forwarding decisions from the exchanged information. Further, this chapter implements Mistill as a PoC and shows for the first time that ML-based TE mechanisms can meet the stringent performance requirements in DCNs.

To show the applicability of Mistill, the forwarding behavior of three TE policies is learned and evaluated in simulations on traffic patterns from four realistic data center applications. The results show that it is possible to learn distributed TE mechanisms from data that closely match the original forwarding decisions and generalize to previously unseen traffic patterns. Further, this chapter analyzes the learned representations and shows that they are reasonable. The PoC shows that Mistill can react at a millisecond scale to changes in the network and that Mistill can be deployed in legacy networks. This chapter thus shows that NN-based TE mechanisms are indeed feasible in practice, opening new avenues of future work in this area.

Chapter 5

Data-driven VNF Assignment Algorithm Design

The virtualization of communication networks and the management of the resulting VNFs, specifically in the context of Cloud-Native Communication Networks [1–3, 9, 254], is a third application area for data-driven algorithm design. The high dynamicity of cloud-native applications makes automation a first principle [255]. Unlocking the full potential of a Cloud-Native Communication Network architecture requires customized resource management algorithms tuned towards particular workload characteristics [2, 256].

Cloud-Native Communication Networks rely on MEC and NFV [2, 254]. MEC provides cloud-computing capabilities at varying distances to the end-users, allowing for low latency and high throughput [7, 257–259]. NFV facilitates packet processing on top of this infrastructure. VNFs running on commodity hardware form the foundation of NFV [11]. Using VNFs leads to easier deployment, maintenance, and improved scalability [11]. Typical VNFs include firewalls, intrusion detection systems, video optimizers, and user plane and control plane functions in mobile networks [1, 26, 260]. VNFs can be composed to arbitrary SFCs [261].

Cloud-Native Communication Networks implement VNFs with micro-services, i.e., decompose a monolithic VNF into smaller CNFs that implement one dedicated task and realize more complex functionality through service-meshes [1, 2, 190, 261, 262]. This architecture enables rapid development, better scalability, maintainability, and deployability compared to monolithic VNFs [255, 263].

Implementing a VNF with CNFs and service-meshes increases the number of deployed CNFs compared to deployments with monolithic VNFs since multiple computationally cheap CNFs implement the monolith's functionality [2, 9, 261]. However, platforms for executing VNFs are ill-suited for CNFs [262]. NF platforms rely on core pinning, i.e., assigning each VNF to a dedicated CPU core [190]. In the context of CNFs, core-pinning can waste resources: A CNF might not fully utilize a core, specifically when processing only a fraction of traffic in multi-tenant environments. Therefore, core-sharing, i.e., assigning multiple CNFs to the same CPU core is desirable [262].

Similar to other placement problems, determining an allocation of CNFs to CPU cores on micro-service-enabled NF platforms requires solving hard combinatorial optimization problems [11, 18]. For example, NF platforms can minimize the maximum load on cores or reduce the number of used cores. The former corresponds to a Load Balancing (LB), the latter to a Bin Packing (BP) problem. Both are NP-complete [16, 264]. In both problems, a solver has to assign N weighted objects on M capacitated bins such that the assignment optimizes the respective objective function. Here, objects correspond to CNFs, bins to CPU cores, an object's weight to the computational demand of a CNF, and a bin's capacity to the computational capability of a CPU core.

Interferences between SFCs, CNFs, and system effects make the computation of a solution even more challenging. For instance, last-level cache interference or the memory layout impacts the computational cost of VNFs [18–20, 190, 265]. Bottlenecking CNFs change the packet rate in SFCs [262]. Varying costs and rates alter the CNFs' demands, which, as this chapter will show, affects how the OS scheduler grants CNFs access to cores in a difficult-to-anticipate way. Testbed measurements will show that including these effects in the assignment is vital to avoid performance degradation, even in lightly loaded scenarios. However, formally modeling these impacts is difficult, time-consuming, and may have to be repeated for different hardware and software stacks [265].

This chapter presents `D2A` (Data-Driven Assignment) that uses Neural Combinatorial Optimization (NCO) paired with Game Theory to learn assignment strategies that incorporate interferences automatically. `D2A` has three advantages: (1) `D2A` learns interference between CNFs; (2) `D2A` tailors solutions to a concrete deployment scenario, e.g., specific SFCs, traffic patterns, or hardware platforms; and (3) `D2A` generalizes to variations in the deployment scenario. To tailor algorithms to CNF interference or hardware effects, `D2A` uses ML to learn a Digital Twin (DT) of the NF platform from generated data and uses the DT to improve learned assignment strategies.

This chapter makes the following contributions:

1. This chapter designs, implements, and evaluates `D2A` to learn data-driven control algorithms for the efficient operation of Cloud-Native Communication Networks.

2. This chapter designs a training method for resource management algorithms by integrating Game Theory and NCO. The training method is applicable beyond `D2A`'s application scenario.

3. This chapter shows the versatility of `D2A` by learning a LB and BP algorithm. `D2A` takes a fully data-driven approach by first learning a DT from data, and then uses DT to learn assignment algorithms. To learn a DT that includes non-linear effects and relational dependencies, `D2A` uses a novel NA using

a Graph Attention Neural Network (GATNN) [266]. The DT can predict throughput and whether cores can handle the load of assigned CNFs.

4. This chapter shows how ML related quality requirements guide the selection of ML models for specific tasks.
5. Through testbed evaluations, this chapter shows that integrating the DT's predictions into the learning process increases throughput compared to an analytical model by 11.38 % and reduces latency by 89.58 %.
6. This chapter provides a comprehensive introduction to the Linux scheduler and unexpected behaviors in the context of co-located CNFs.
7. The release of data [267] to foster reproducibility and support the community in the research of interference of co-located CNFs.

In summary, this chapter provides valuable information for the operation of Cloud-Native Communication Network.

Organization. This chapter is organized as follows: Sec. 5.1 introduces background information and related work. Sec. 5.2 is partially based on [29, 268] and describes the business objective, and operational requirements towards the control algorithm, formally models the underlying optimization problem, and explains peculiarities of co-locating CNFs on one core and how those peculiarities affect throughput and latency. Sec. 5.3 describes the operational requirements, design of `D2A` , and the generation of a challenging workload. Sec. 5.4 is partially based on [29] and describes how `D2A` integrates Game Theory, NCO, and DTs into a framework that simplifies the learning of control algorithms into the design of a game. Sec. 5.5 is based on [29] and describes the DTs that guides the learning. Sec. 5.6 is based on [29] and compares measurement results of `D2A` and baseline algorithms and the predictive performance of DTs. Finally, Sec. 5.7 summarizes and concludes this chapter.

5.1 Background and related work

This section introduces background information on the `OpenNetVM` NFV platform in Sec. 5.1.1, process scheduling in Linux in Sec. 5.1.2, background on CNF scheduling in Sec 5.1.3, Game Theory in Sec. 5.1.4, NCO and RL in Sec. 5.1.5, and gives on overview over related work and the SoA in Sec. 5.1.6. Specifically, Sec. 5.1.2 and Sec 5.1.3 correspond to MaLANe's *Analyze Networked System* activity. Sec. 5.1.4 and Sec. 5.1.5 reflect MaLANe's *Formalize Problem* activity.

5.1.1 The `OpenNetVM` platform

Fig. 5.1 shows the architecture of `OpenNetVM` [269], which forms the basis of `D2A`. `OpenNetVM` is based on the Data Plane Development Kit (`DPDK`) [270] and

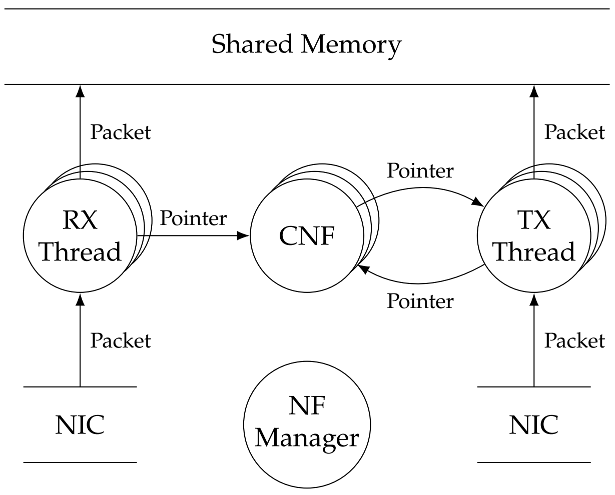


Figure 5.1 `OpenNetVM` system architecture. The NF Manager creates a shared memory region to store packets and metadata such as the flow table and SFC lists. Packets are moved between NFs by RX and TX-threads that copy packet descriptors into an NF's receive (R) and transmit (T) ring buffers. NFs run in isolated containers that encapsulate all dependencies.

designed for service function chaining. `OpenNetVM` provides zero-copying of packets between VNFs, setting `OpenNetVM` apart from other systems in the line of `ZygOS` [271], which focus on a single application.

The most important entities of `OpenNetVM` are the RX- and TX-Threads, the `NFManager`, and containerized NFs. Upon startup, the `NFManager` creates a shared memory region for all NFs. The `NFManager` spawns TX- and RX-threads. The RX-thread fetches packets from the NIC, places the packets in the shared memory region, and inserts pointers into the input buffers of the first NFs in the configured SFCs. The TX-threads copy pointers between the output and input buffers of chained NFs. The TX-threads also transfer outbound packets to the corresponding NICs. `OpenNetVM` uses core-pinning, i.e., assigns each VNF to a dedicated core. Core-pinning can result in significant resource overhead if individual VNFs do not utilize their CPU cores fully. Underutilization can occur if micro-service architectures are adopted for VNFs [262], or in multi-tenant systems where individual SFCs serve only a small fraction of the overall traffic [190].

5.1.2 Scheduling in Linux

`OpenNetVM`, similar to other NF platforms, runs on top of Linux [190, 262, 269, 272, 273]. Since the chapter's objective is the development of a data-driven assignment algorithm for CNFs that are co-located on CPU cores, understanding how Linux grants processor time to co-located threads is important. This knowledge will form the basis of the learning task and is necessary to understand the peculiar behavior of the system.

Linux offers two primary scheduling policies: real-time scheduling and the Completly Fair Scheduler (CFS) [274–276]. This section focuses on the CFS, since

DPDK, forming the basis of OpenNetVM, forbids using real-time policies in combination with the `rte_ring` structure, since the real-time policy in combination with `rte_ring` can lead to deadlocks [277]. Testbed experiments with the real-time policies and OpenNetVM always resulted in a system freeze. OpenNetVM would have to be rewritten and use the lock-free stack mempool handler to use real-time policies [277].

The Completely Fair Scheduler

Every operating system has a (process) scheduler that organizes the access of threads in the runnable state[1], denoted by $\mathcal{E}$ to a CPU core. The scheduler decides how the finite processing time of the CPU is divided among runnable threads. On an ideal processor, each thread would receive a fraction of $\frac{1}{|\mathcal{E}|}$ of the processor's time. The time is inversely proportional to the number of runnable threads. To achieve this, the ideal processor would schedule each process for an infinitely small duration. With the CFS, Linux imitates this behavior.

Time slicing in the CFS

The CFS does not assign a fixed time slice for each runnable thread. Similar to the ideal scheduler, the CFS calculates a thread's time on the CPU as a function of the total number of runnable threads. To calculate this time slice, the CFS uses a time interval called *target latency* Δt_l[2]. Each runnable thread gets a time slice of:

$$\text{ts} : \mathcal{E} \to \mathbb{R}_{>0}, e \mapsto \frac{\Delta t_l}{\mid \mathcal{E} \mid}. \tag{5.1}$$

The target latency approximates the infinitely small switching time of the ideal processor [275, Chapter 4].

Eq. (5.1) shows that each thread's time slice approaches zero as the number of runnable threads approaches infinity. Since switching between threads incurs an overhead, the CFS defines a minimum value for the duration of the time slice, the *minimum granularity* Δt_g[3] [275, Chapter 4]. Eq. (5.1) thus becomes:

$$\text{ts} : \mathcal{E} \to \mathbb{R}_{>0}, e \mapsto \min\left(\frac{\Delta t_l}{\mid \mathcal{E} \mid}, \Delta t_g\right). \tag{5.2}$$

Consequently, each thread will run for at least Δt_g ms no matter how many runnable threads want access to the CPU.

Calculating each thread's time slice with Eq. (5.2) grants each thread the same time on the CPU. However, a user might be more interested in the results of one program than another. For example, when playing a game, a user might prefer

[1]The term *runnable* refers to a well-defined state in the life cycle of a process and identifies threads that are ready for execution on the CPU [276, Chapter 9.4].
[2]The default target latency on, e.g., Ubuntu 22.04 is $\Delta t_l = 24\,\text{ms}$.
[3]The default minimum granularity on, e.g., Ubuntu 22.04 is $\Delta t_g = 3\,\text{ms}$.

a fluid rendering over a backup job running in the background. If necessary, the user would assign the thread(s) of the game a higher priority than the backup job, i.e., the user would want the thread(s) of the game to receive more time on the CPU. The CFS allows this through so-called *Nice* values taking a value in $\{-20, -19, \ldots, 19\}$. Formally, this chapter defines the Nice value of a thread as a function:

$$\mathrm{n} : \mathcal{E} \to \{-20, -19, \ldots, 19\}, \tag{5.3}$$

A Nice value of -20 corresponds to the highest priority, a nice value of 19 to the lowest priority, and the default value is zero [275, Chapter 4].

The CFS incorporates the Nice values into the calculation of the target latencies by weighting the sum in Eq. (5.2), thus Eq. (5.2) becomes:

$$\mathrm{ts} : \mathcal{E} \to \mathbb{R}_{>0}, e \mapsto \min\left(\frac{\mathrm{w}\left(\mathrm{n}\left(e\right)\right) \Delta t_l}{\sum_{b \in \mathcal{E}} \mathrm{w}\left(\mathrm{n}\left(b\right)\right)}, \Delta t_g\right), \tag{5.4}$$

where the function $\mathrm{w} : \{-20, 19, \ldots, 19\} \to \mathbb{N}$ maps the Nice value to the actual weights [278, Version 6.0.10, `kernel/sched/core.c`, Line 11202ff]. Once the time slice $\mathrm{ts}\left(e\right)$ of a thread $e \in \mathcal{E}$ expires, the CFS preempts the thread and picks a new thread for running on the CPU.

Enforcing time slices in practice

Being able to calculate a thread's time slice leads to a new problem: The enforcement of this time slice, i.e., ensuring that the CFS correctly preempts the task. To understand how the Linux kernel does this, understanding the concept of time in the kernel is crucial. The kernel measures time through a system timer based, e.g., on the processor's frequency. The system timer issues an interrupt at a specific tick rate. The kernel stores the tick rate in the `Hz` variable[4]. A special interrupt handler in the kernel handles this interrupt. One task in this interrupt handler is updating the runtime of the currently running thread [275, Chapter 11]. As a result, the accounting of runtime is imprecise. For example, if the time slice of a thread is just about to expire and the interrupt is invoked, the thread remains on the CPU until the next interrupt, thus gaining up to almost 4 ms of extra time on the CPU. On the long run, this imprecision averages out since it applies equally to all threads. Still, this imprecision is undesired and should be minimized. Intuitively, the next time the thread runs, the CFS should shorten the thread's time slice accordingly. To do that, the CFS needs to track the execution time of each thread on the CPU. The CFS keeps track of the wall-clock time a thread spends on the CPU and maintains a separate *virtual runtime*, `vruntime` for short. The CFS uses the `vruntime` for scheduling decisions.

[4]The default value of the `Hz` variable is 250 Hz on Ubuntu 22.04. Thus, the interrupt is invoked every 4 ms.

During the interrupt handler, the kernel calculates the currently running thread's `vruntime` increment according to Eq. (5.5):

$$\mathrm{vinc} : \mathcal{E} \times \mathbb{N} \to \mathbb{N}, (e, d) \mapsto d\frac{\mathrm{w}(0)}{\mathrm{w}(\mathrm{n}(e))}, \tag{5.5}$$

where d corresponds to the time thread e spend running on the CPU [278, Version 6.0.10, `kernel/sched/fair.c`, Line 297ff]. Eq. (5.5) shows that the CFS weights the execution time relative to the Nice value zero. Thus, the `vruntime` of threads with a Nice value of zero corresponds to the actual wall-clock time. For a thread, e with $\mathrm{n}(e) > 0$, i.e., a thread with a lower priority and thus lower weight, the `vruntime` increases faster than wall-clock time. For a thread e with $\mathrm{n}(e) < 0$, i.e., a process with a higher priority and thus larger weight, the `vruntime` increases slower than the wall-clock time.

After updating the `vruntime` of the currently running thread e, the CFS checks if there exists any runnable thread g whose `vruntime` is smaller than e's `vruntime` plus the minimum granularity Δt_g. If such a thread exists, then the CFS changes e to the runnable state, and g to the running state, i.e., g executes on the CPU. For efficiency, the kernel organizes the threads in a red-black tree with the `vruntime` as key [275, Chapter 4].

As a consequence of this procedure, the CFS never has to calculate a thread's time slice explicitly. Instead, the CFS uses the `vruntime` to elegantly realize the time slices according to Eq. (5.4) while at the same time accounting for timing imprecision.

Fine-grained scheduling with control groups

Sometimes, the control that the CFS provides is insufficient. One such example is systems that serve multiple users. Intuitively, the system should grant both users the same amount of CPU time. With the CFS, both user sessions would require running the same number of threads. If this is not the case, e.g., if one user has twice the number of threads than the other, then this user would get twice as much CPU time. Adjusting the thread's Nice values to correct this imbalance is possible but would require readjusting the Nice values every time the number of threads changes. Instead, the Linux kernel provides so-called control groups (`cgroups`).

Introducing `cgroups`. `cgroups` are a powerful tool for fine-grained control of resources such as CPU, memory, disk, network, etc. A `cgroup` is a collection of threads bound to a set of limits. Further, `cgroups` allow the hierarchical organization of resources [279]. CPU `cgroups` allow the control to the CPU via a *weight* parameter, often referred to as *shares*[5]. The weight parameter takes a

[5]In the first version of the `cgroups` control system, this parameter was named *shares*.

value in the set $\{1, 2, \ldots, 10\,000\}$, and defaults to 100 [280]. The absolute value of the parameter has no meaning. The weight parameter allows a fine-grained distribution of CPU time to `cgroups` according to the same principle as in Eq. (5.4). Let C bet the set of `cgroups` for the CPU resource, and $\mathrm{s} : C \to \{1, \ldots, 10\,000\}$ a function that maps a `cgroup` to its weight. Then, a `cgroup` $c \in C$ gets a fraction:

$$\frac{\mathrm{s}(c)}{\sum_{i \in C} \mathrm{s}(i)} \tag{5.6}$$

of the CPU time. The threads in each `cgroup` then divide the `cgroup`'s CPU time among themselves according to Eq. (5.4) [280, 281]. Note that each thread's Nice value impacts its importance relative to the threads in its `cgroup`, *but not to threads in other `cgroups`* [279].

With `cgroups`, ensuring an equal share of CPU time for each of the two users in the earlier example thus amounts to grouping each user's threads into one `cgroup`, and assigning each `cgroup` the same weight. `cgroups` also form the basis of container technologies, i.e., threads related to containers are isolated from each other through `cgroups` [262]. Container resource utilization can then be controlled through the `cgroup` interface. `cgroups` are thus a crucial building block of Cloud-Native Communication Network.

`cgroups` and the CFS. The Kernel elegantly includes the `cgroups` into the CFS through a composite design pattern [278, Version 6.0.10, `include/linux/sched.h`, Line 547ff][281]. The `cgroups` are the composite objects, and the individual threads are the Leaf objects. Each component tracks the `vruntime` and the wall-clock execution time. Composite objects store their children in a red-black tree keyed by the `vruntime` variable. Children update the `vruntime` and the execution time of their parents if their values change. For this, Eq. (5.5) is updated to:

$$\mathrm{vinc} : \mathcal{E} \cup C \times \mathbb{N} \to \mathbb{N}, (e, d) \mapsto \begin{cases} d\frac{w(0)}{w(n(e))} & \text{if } e \in \mathcal{E} \\ d\frac{s(e)}{\sum_{i \in C} s(i)} & \text{else.} \end{cases} \tag{5.7}$$

Thus, the CFS can start at the root object and then recursively traverse the hierarchy, always picking the leftmost element in the red-black tree, until the CFS encounters a leaf object, i.e., the thread that it then runs on the CPU [281].

5.1.3 Towards fair µVNF scheduling

In the networking context, the CFS's concept of fairness can result in packet loss and rate warping. For example, two CNFs, `NF1` and `NF2` share one CPU core. For `NF1`, 1 % of CPU time is enough to process the arriving packets. In contrast, `NF2` is more expensive and requires 70 % of CPU time. In total, the two CNFs need 71 % of CPU time. Still, `NF2` will drop packets, since the CFS would grant `NF2` only 50 % of CPU time. What's worse, `NF2` has to wait until `NF1`'s `vruntime` exceeds its own. Since `NF1` requires only a small processing time, a substantial amount of time might be necessary until the CFS lets `NF2` run again. This, in turn, can lead to a substantially higher decrease in throughput than the expected 20 %.

To operate well in the context of networking, the CFS needs another concept of fairness: rate-cost proportional fairness (`RC`), i.e., dividing CPU time proportional to the product of rate and compute demand of co-located CNFs [262]. Let $\mathcal{J}$ be the set of CNFs co-located on one CPU core. Further, assume that each CNF is grouped into its own `cgroup`, and the function $\mathrm{cg} : \mathcal{J} \rightarrow C$ returns the `cgroup` of a CNF. Let $\lambda_{\mathrm{a}} : \mathcal{J} \rightarrow \mathbb{N}$ be the packet arrival rate, and $\mathrm{c} : \mathcal{J} \rightarrow \mathbb{R}_{>0}$ be the average number of CPU cycles to process one packet of a CNF. Then the weight parameter of the CPU `cgroup` $\mathrm{cg}(j)$ of CNF j should be set to:

$$\mathrm{s}\left(\mathrm{cg}(j)\right) = \min\left(\left\lceil 10\,000 \frac{\lambda_{\mathrm{a}}(j)\,\mathrm{c}(j)}{\sum_{i \in \mathcal{J}} \lambda_{\mathrm{a}}(i)\,\mathrm{c}(i)} \right\rceil, 10\,000\right). \tag{5.8}$$

This ensures that each CNF gets CPU time proportional to its rate-cost product, eliminating throughput degradation as in the earlier example. In addition, `RC` ensures that rates reduce equally in the case of overload, i.e., the scarce resources are distributed so that the throughput of affected flows behaves as it would on a bottlenecked link [262].

5.1.4 Game Theory

This section introduces basic concepts from game theory that will become relevant later in the chapter. This section first introduces games in their normal form and defines the Nash Equilibrium (NE). Then, this chapter introduces sequential games, which form the basis of Sec. 5.4.1. In the scope of sequential games, this section then introduces backward induction and generalizes the concept of NE to the Subgame-perfect Nash Equilibrium (SNE). Large parts of this section are based on Gibbson [282], and Nisan et al. [283].

Normal-form games

A game in normal form is a tuple $G = (\mathcal{J}, \mathcal{S}, \mathrm{u_n})$. The set $\mathcal{J}$ corresponds to the players in the game. The strategyspace $\mathcal{S} := \times_{j \in \mathcal{J}} \mathcal{S}_j$ corresponds to all possible combinations of individual player strategies $\mathcal{S}_j$. An element $S \in \mathcal{S}$ is called a strategy profile. The utility function $\mathrm{u_n} : \mathcal{J} \times \mathcal{S} \rightarrow \mathbb{R}$ returns the utility of a player's action given the actions of all other players [283]. In a normal-form game, the players move simultaneously, i.e., each player chooses her action without knowing the other player's actions [282, 283].

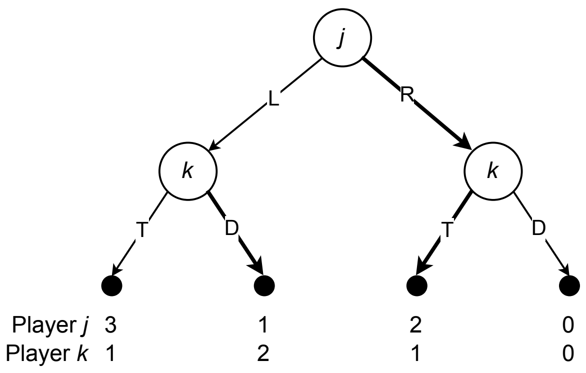


Figure 5.2 Example of a game in extensive form. The players are $\mathcal{J} = \{j, k\}$. The moves for each player are $\mathcal{A}_j = \{L, R\}$, and $\mathcal{A}_k = \{T, D\}$. The game has two stages. First, player j makes her move. Player k observes player j's move and makes her move, and the game ends. The figure shows the player's utility value for each possible outcome below the leaf nodes. The figure indicates the SNE with bold arrows. Figure adapted from Gibbson [282].

Nash Equilibrium

A NE $S^* \in \mathcal{S}$ corresponds to a strategy profile in which no player j has the incentive to deviate from her chosen strategy s_j^*. Formally [282, 283]:

$$\forall j \in \mathcal{J} : \forall S \in \mathcal{S} \setminus S^* : \quad \mathrm{u_n}(j, S^*) \geq \mathrm{u_n}(j, S) . \tag{5.9}$$

Consequently, the strategy s_j^* of player j is a best response to all other player strategies. The strategy s_j^* is a best response iff [282, 283]:

$$s_j^* = \max_{s_j \in \mathcal{S}_j} \mathrm{u_n}\left(j, \left(s_1^*, \ldots, s_j, \ldots s_{|\mathcal{J}|}^*\right)\right) . \tag{5.10}$$

The concept of a NE assumes that all players are rational, i.e., choose the strategy that maximizes their utility.

Normal-form games can have two types of NE: pure and mixed strategy NEs. All players choose one specific strategy in a pure-action NE. In a mixed strategy NE, players choose their strategy according to a probability distribution that depends on the utility structure of the game. Every normal-form game has at least one mixed strategy NE. Normal-form games are not guaranteed to have a pure strategy NE [282].

Extensive-form games

For games in which players move in sequence, i.e., in which the assumption of simultaneously choosing a strategy does not hold, the representation in extensive form is more convenient. An extensive-form game specifies five elements [282]:

1. The players in the game, i.e., $\mathcal{J}$,
2. when each player has to move,

Player *k*

Player *j*		T-T	T-D	D-T	D-D
	L	3,1	3,1	1,2	**1,2**
	R	2,1	0,0	**2,1**	0,0

Figure 5.3 Example of a game in normal form. The game corresponds to the extensive-form game in Fig. 5.2. The strategies of each player are $\mathcal{S}_j = \{L, R\}$, and $\mathcal{S}_j = \{T-T, T-D, D-T, D-D\}$. The strategies of player k indicate the move for each of player j's moves. For example, $T-D$ indicates that player k makes move T if player j makes move L, and player k makes move D if player j makes move R. Each player's utility for a strategy profile is given in the cells. The first number corresponds to the utility of player j, and the second number to player k's utility. NEs are indicated in a bold font face. Figure adapted from Gibbson [282].

3. player j's available actions $\mathcal{A}_j$ each time the player moves,
4. the player's knowledge, i.e., what information the player has available each time the player moves,
5. the utility $\mathrm{u_e}$ received by each player for each combination of moves.

Extensive-form games can be represented as normal-form games and vice-versa. The distinction between a move in the extensive-form game and a strategy in the normal-form game is subtle and will be explained shortly.

Extensive-form games are often represented as a game tree. Fig. 5.2 shows a game tree adapted from [282]. Nodes in the tree correspond to decision points at which a player has to make her move. The node is labeled with the corresponding player's identifier. Arcs between nodes correspond to a move a player has made. The arcs are labeled with an identifier for this move.

The difference between a move and a strategy in the normal-form game is that the strategy describes a full plan of action, i.e., specifies all moves in every possible contingency a player might have to act [282]. In the above example, a strategy of player k has to specify the movements of k for every move that j makes. Thus, k has four strategies in the game in Fig. 5.2. Fig. 5.3 shows the corresponding game in normal form.

Backwards Induction

Backward Induction is a technique to specify the player's moves in extensive form games with complete and perfect information. A game has complete information if each player's utility function is common knowledge. A game has perfect information if every time a player has to move, all previous player's moves are common knowledge. Then, each player's utility can be computed for all leaves in the game tree, i.e., for all possible combinations of moves. At every intermediate node, the player to move at this node can thus compute the move that maximizes its utility [282].

For example, player k in Fig. 5.2 can compute the best move to every given move a_1 of player j:

$$R(k, a_j) = \max_{a_k \in \mathcal{A}_k} \mathrm{u_e}\left(k, \left(a_j, a_k\right)\right). \tag{5.11}$$

Similarly, player j can compute its move as:

$$\max_{a_j \in \mathcal{A}_j} \mathrm{u_e}\left(j, \left(a_j, R(k, a_j)\right)\right). \tag{5.12}$$

As the next section shows, the Backward Induction outcome is closely related to the concept of NE in the normal form game.

Subgame-Perfect Nash Equilibrium

Before generalizing the NE to extensive-form games of complete and perfect information, a subgame must be introduced. A subgame is part of an extensive-form game. The subgame starts at an arbitrary node and contains all subsequent nodes and leaves of the game tree. A subgame does not contain the full extensive-form game, i.e., cannot start at the root node [282]. A SNE is a strategy profile that corresponds to an NE in every subgame [282].

For example, the SNE for the game in Fig. 5.2 is (R, TD). The strategy of player j is R. The strategy of player k is TD, i.e., if player j chooses L, then player k would choose T, and if player j plays R, player k plays D. In contrast, the backward induction outcome corresponds to $a_j = R$, and $a_k = T$.

Comapred to the extensive-form game in Fig. 5.2, the normal-form representation in Fig. 5.3 has two NE: (L, DD) and (L, TD). However, (L, DD) is no SNE since it is not a NE in the subgame starting at the decision node of player k after player j made a move R. Here, the strategy DD would result in a zero payoff for player k. Instead, a rational player would make a move T at this decision point. Thus, the strategy profile (L, DD) is not a NE in this subgame, violating the definition.

5.1.5 Reinforcement Learning and Neural Combinatorial Optimization

This section introduces NCO. Since NCO relies on RL, this section starts by introducing RL with a focus on discrete action spaces and finite episodes, i.e., the RL problem ends after a certain number of steps. Large parts of this section are based on Sutton and Barto [81].

Single-Agent RL.

Fig. 5.4 shows the basic setting for single-agent RL. A single decision-maker, an agent, is situated in an environment with unknown dynamics. The agent's goal is to optimize numerical feedback, called reward, that the agent receives while interacting with the environment. Interactions between the agent and

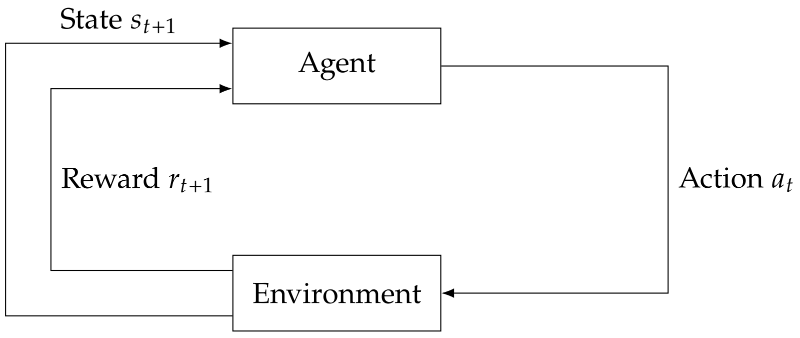


Figure 5.4 The agent-environment interface. The agent chooses an action a_t and executes it in the environment. The environment reacts to the action and emits a new state, i.e., observation s_{t+1}, together with a reward signal r_{t+1}. The agent observes both.

environment occur in discrete time steps through actions. At each time step, the agent executes an action a_t. The action results in a change of the environment, which transitions from its current state s_t to a new state s_{t+1}. As a result of this transition, the agent observes the numerical reward r_{t+1}. The agent's goal is the optimization of the reward. Further, the agent observes the new environment state s_{t+1}. Formally, this process is modeled through a Markov Decision Process (MDP) [81, 284].

The Markov Decision Process (MDP) A MDP is a tuple $(\mathcal{A}, \mathcal{S}, p, \mathrm{r})$. The finite action space $\mathcal{A}$ contains the actions available to the agent at each time step. The state space $\mathcal{S}$ contains all environment states. The function $\mathrm{p} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ represents the environment dynamics, i.e., the probability of transitioning into state s_{t+1}, given that the agent applies action a_t while the environment is in state s_t. The reward function $\mathrm{r} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ returns the numerical reward for the transition from state s_t to s_{t+1} given that the agent chose action a_t. To optimize the reward, the agent has to learn how to act, i.e., the agent must learn a behavioral policy [81, 284].

The policy function Formally, a policy is defined as:

$$\pi : \mathcal{A} \times \mathcal{S} \rightarrow [0, 1], \tag{5.13}$$

i.e., the probability of choosing an action $a \in \mathcal{A}$, given that the environment is in the state $s \in \mathcal{S}$ [81]. Further, RL usually involves the estimation of a *value* or *action-value* function. Those functions estimate the agent's utility of being in a certain state or choosing a specific action in a specific state.

The value and action-value function Formally, the value function can be defined as:

$$\mathrm{v}_\pi : \mathcal{S} \rightarrow \mathbb{R}, s \mapsto \mathbb{E}_{s_{t+1} \sim \mathrm{p}, a_t \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \, \mathrm{r}\left(s_t, s_{t+1}, a_t\right) \mid s_0 = s \right], \tag{5.14}$$

where $\gamma \in [0, 1)$ is a discount factor. The value $\mathrm{v}_\pi(s)$ of a state $s \in \mathcal{S}$ is the expected discounted reward when starting in the state s. Following the policy, π thereafter, i.e., subsequent states are sampled according to the environment dynamics, and actions according to π [81, 284].

Similarly, the action-value function can be defined as:

$$\mathrm{q}_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}, (s, a) \mapsto E_{s_{t+1} \sim \mathrm{p}, a_{t>0} \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \, \mathrm{r}\,(s_t, s_{t+1}, a_t) \mid s_0 = s, a_0 = a \right], \tag{5.15}$$

where $\gamma \in [0, 1)$ is a discount factor. The action-value $\mathrm{q}_\pi(s, a)$ of taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ is the expected discounted reward of taking action a in state s and following the policy π thereafter[6].

Neural Combinatorial Optimization

NCO is a recent trend to solve COP with the help of ML. The seminal paper by Bello et al. [285] proposes NCO as solving COPs with RL and NNs.

NCO maps a COP into a MDP and then applies RL to learn a policy that produces solutions to the COP. Optimizing the policy with RL then corresponds to learning an algorithm that finds better and better solutions to the COP [285].

To cope with the combinatorial search spaces of COPs, NCO relies on NNs. The NN thereby represents, depending on the chosen RL technique, the policy, value, or action-value function [284].

Intuitively, NCO offers the possibility to automate the design of heuristic algorithms in the form of NNs that produce solution instances to a particular set of COP problems. NCO does not replace conventional solvers that guarantee optimal solutions. The learned algorithms can be expected to work well on problem instances similar to the ones seen during training but might be arbitrarily bad when applied to samples that differ strongly [28, 81, 284].

5.1.6 Related Work

This section introduces related work categorized into six classes: VNF placement, performance modeling, thread scheduling, NCO, NF platforms, and dataplane operating systems.

VNF placement

Most prior work focuses on placing VNFs *on servers* [11]. Instead, the work in this chapter focuses on assigning CNFs *to CPU cores*. Most related is the work of Wang et al. [135] and Zheng et al. [18]. Both minimize data transfer between VNFs in

[6]Eq. (5.14) and Eq. (5.15) use an infinite number of steps to better align with related work and simplify the notation. Episodic RL problems can be converted easily to the infinite formulation [81].

the same SFC on different Non-Uniform Memory Access (NUMA) nodes; both perform core pinning.

In contrast, the work in this chapter targets systems that allow core sharing. Also, the proposed framework can be applied to arbitrary NF platforms, whereas results in [135] and [18] are limited to specific hardware settings. The framework proposed in this chapter is data-driven and thus not bound to a specific hardware setting. Instead, the framework is designed to adapt to changing configurations automatically.

Performance modeling

The memory subsystem, data transfer between NUMA nodes, and interference on the NIC can impact the throughput, latency and, CPU usage of NF systems [19, 20, 265, 286, 287]. Interference and computational cost can be modeled with ML [265, 288].

This chapter adds to potential sources of interference by explaining the impact of the CFS on the packet processing of co-located CNFs. A new aspect that has not been reported previously.

Further, this chapter complements previous modeling approaches with a NA for arbitrary hardware and problems. The focus of the NA is the prediction of VNF throughput and core tipping, but it can, in principle, be used to learn arbitrary aspects of VNF platforms. The problem representation proposed in Sec. 5.5.2 easily extends to arbitrary metrics on the VNF, SFC, CPU, NIC, and node level.

Thread scheduling

The work in this chapter is closely related to thread scheduling [289–292]. Similar to OS schedulers, those algorithms are not designed with the specific workload of network processing in mind. The above solutions rely on the frequent migration of threads between cores to improve performance metrics. However, frequent migration of VNFs is harmful for the specific networking workload [18]. Work in this chapter, therefore, focuses on finding assignments of CNFs to cores that allow the reliable processing of packets over longer periods.

Neural Combinatorial Optimization.

NCO, i.e., solving combinatorial optimization problems with RL has become an active area of research in the algorithmic [17, 293], ML [28], and networking community [15]. Particularly related is related work that learns the admission of jobs [294] and the assignment of jobs to computing resources [295–298].

This chapter complements previous work in the area by taking a fully data-driven approach. The proposed framework uses NCO to learn a data-driven heuristic to a combinatorial optimization problem and further uses ML to learn environmental effects and include those into the learned heuristic. Further, this work integrates Game Theory into NCO to empower the resulting data-driven heuristic with higher flexibility.

NFV platforms.

A range of NF platforms and systems exist [190, 262, 269, 272, 273, 299–302]. Work in this chapter extends `OpenNetVM` [269] and allows sharing of CPU cores between CNFs as in [262]. The work in this chapter is orthogonal to previous work in that this chapter focuses on how to assign VNFs to CPU cores to improve overall system performance. In contrast, previous work on the co-location of CNFs takes the assignment as given. While the proposed framework is implemented on top of `OpenNetVM`, the proposed framework can be combined with any NF platform to improve the underlying assignment of CNFs to CPU cores.

Dataplane operating systems.

The objective of Dataplane operating systems [271, 303–306] is to provide microsecond tail latencies for data center applications. They are not designed for the SFC use-case, e.g., they do not support zero-copying of packets between VNFs as `OpenNetVM` does. Techniques used in those systems could be combined with existing NF platforms, though. Further, the goal of $\texttt{D}_2\texttt{A}$ is the co-location of CNFs such that overall system performance is improved, which dataplane operating systems do not consider.

5.2 System analysis and problem formulation

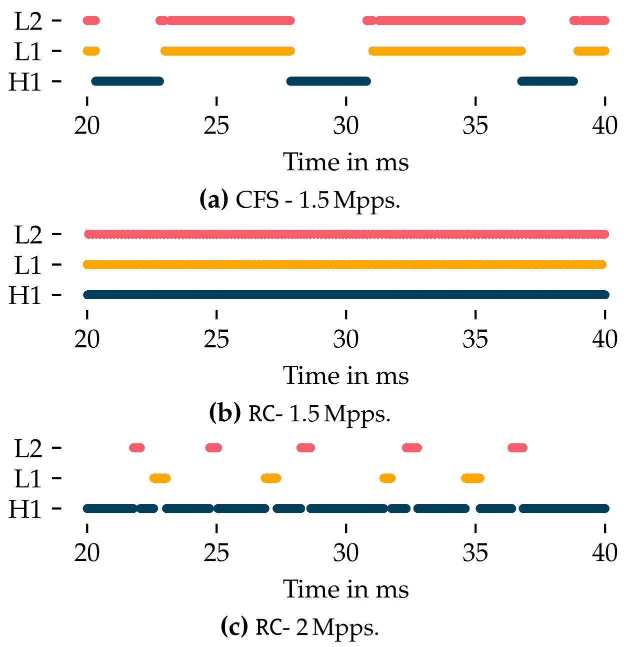
This section introduces and analyzes Linux scheduling policies in Sec. 5.2.1, formulates the operational objectives towards a cloud-native ready NF platform in Sec. 5.2.2, and translates and formulates the resulting assignment problem formally in Sec. 5.2.3. Thus, this section contains aspects of MaLANe's *Translate Objective*, *Formalize Problem*, and *Analyze Networked System* activities.

5.2.1 Process scheduling in `OpenNetVM`

The NF platform's underlying OS manages the access of co-located NFs to the respective CPU core, which is the task of the OS 's CPU scheduler. The scheduler ensures that each process assigned to a core gets processing time and can forcibly interrupt a process if the process runs for too long.

Default Linux scheduling

NF platforms usually run on top of Linux [190, 262, 269, 272, 273]. The Linux kernel offers two scheduling policies: Real Time (RT) and non-RT policies [274, 275]. This chapter does not consider RT policies because `DPDK` forbids their



(a) CFS - 1.5 Mpps.

(b) RC- 1.5 Mpps.

(c) RC- 2 Mpps.

Figure 5.5 Activity of a heavy and two light CNFs with the normal CFS scheduler (see (a)) and the RC scheduler (see (b) and (c)). Figures (a) and (b) show how weighting affects the execution of the CNFs. Figure (c) shows how a slight packet arrival rate increase results in a new mode of operation.

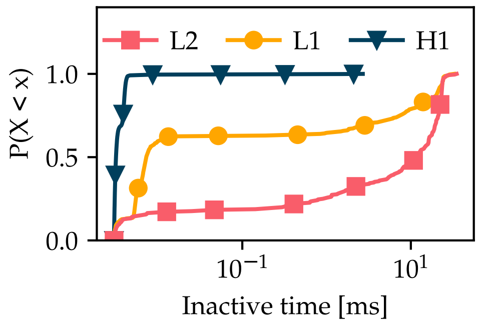


Figure 5.6 The CDFs show the CNFs' inactive times from Fig. 5.5c for the whole experiment duration.

usage in combination with the `rte_ring` structure [7]. The non-RT policy extends the CFS with two configurations: `SCHED_NORMAL` and `SCHED_BATCH` [274]. `SCHED_NORMAL` is the default configuration and handles interactive and background tasks. `SCHED_BATCH` tunes the behavior of the CFS towards compute-intensive tasks and applies a small scheduling penalty concerning the wakeup behavior [274]. Thus, `SCHED_BATCH` has little impact in the considered scenarios, and measurements are obtained with `SCHED_NORMAL.`

Fig. 5.5a illustrates the behavior of the CFS with three CNFs assigned to one core. Each CNF has an arrival rate of 1.5 Mpps. The CNFs L1 and L2 have a computational cost of 210 Cycles per packet (Cpp). The CNF H1 has a cost of 1 600 Cpp. The CNFs yield if their buffer is empty. Fig. 5.5a shows that H1 is active for 2.5 ms, and L1 and L2 take turns for 6 ms. Thus, H1 is inactive for 6 ms, and L1 and L2 are inactive for 2.5 ms. The CFS's `vruntime` accounting and job preemption cause this. The Kernel checks every 4 ms if it should preempt a CNF[8]. Until then, the CNF runs on the core. In Fig. 5.5a, H1 uses the available time and does not voluntarily yield, increasing its `vruntime` accordingly. L1 and L2 are faster and yield, and thus take turns: L2 processes a few packets, yields, and the Kernel increments the `vruntime`. L1 has the lowest `vruntime` now, and the Kernel schedules it to the core. L1 processes packets and yields, and the Kernel increases the `vruntime`. This process repeats until the `vruntime` of L1 and L2 exceeds the `vruntime` of H1. The frequent switching adds overhead due to context switches, resulting in the 6 ms duration. The behavior of the CNFs results in jitter and latencies in the order of milliseconds. Further, the CPU time allocation is unfair concerning the rate-cost product[9]. All three CNFs get the same CPU time, although the rate-cost product of H1 is eight times larger than that of L1 and L2.

Rate-cost proportional fairness

Fig. 5.5b shows the behavior of `RC`. The Nice values are set to achieve rate-cost proportional fairness. As a result, the `vruntime` of L1 and L2 increases faster than the `vruntime` of H1. Thus, L1's and L2's `vruntime` quickly exceeds H1's `vruntime`. Therefore, the Kernel schedules H1 more often, reducing its buffer and the time H1 needs until it can yield.

[7]RT policies in combination with `rte_ring` can lead to deadlocks [277]. Testbed experiments with the real-time policies and `OpenNetVM` always resulted in a system freeze. `OpenNetVM` would have to be rewritten and use the lock-free stack `mempool` handler to use real-time policies.

[8]The value of 4 ms is the frequency of the system timer, stored in the Kernel's `Hz` variable. The system is configured with 250 Hz. Values between 100 Hz to 1 000 Hz are recommended. Changing the `Hz` variable requires a re-compilation of the Kernel [275].

[9]The rate-cost product for an NF is the product of the NF's processing cost per packet and packet arrival rate.

Core-tipping: RC is not enough

Even with RC, the CFS′ behavior can be unexpected. Fig. 5.5c increases the packet arrival rate from 1.5 Mpps to 2 Mpps; the NFs′ Cpp values are unchanged, resulting in the same weights for RC. Fig. 5.5c shows a different behavior than Fig. 5.5b. H1 blocks the core in the order of milliseconds. Further, the NFs′ inactive times are not deterministic. To better understand, Fig. 5.6 shows a CDF of the NFs′ inactive times for the whole experiment period. L1 and L2 are inactive for up to 12 ms, resulting in high latencies and substantial jitter.

Incorporating this behavior into the assignment decision is thus of high importance. However, the tipping point at which the system breaks requires solving a complex dynamical problem that depends on starting conditions such as the CFS and Kernel parameterization, the cost and rates of CNF, and their variance. Fluctuating processing costs, bursty packet arrivals, and inter-dependencies between CNFs in the same SFC assigned to different cores further complicate the problem. Since this behavior can occur even on seemingly underutilized cores, we refer to this behavior as a core-tip. This thesis shows that data-driven algorithms can learn when a core-tips and thus improve performance.

5.2.2 Operational objectives

For a NF platform, three KPIs are identified:

1. Throughput: The number of packets the platform can process per second.
2. Latency: The per-packet processing latency.
3. Total Packet Cost (TPC): The ratio of required CPU cycles to the number of processed packets.

Throughput and latency are customer-facing KPIs. Throughput and latency are essential KPIs in many applications that run on top of communication networks. For example, 5G use-cases such as the tactile Internet frequently require low end-to-end latencies [6]. End-users usually have a covenanted minimum throughput that the service provider has to ensure by law [307, Vfg Nr. 99/2021].

TPC is tied to the network operator and measures the infrastructure efficiency. One goal of Cloud-Native Communication Networks is increasing the efficiency of the infrastructure by scaling CNFs to the demand. That is, a Cloud-Native Communication Network does not provision CNFs for peak demand but can scale them horizontally or vertically to consume more resources during peak hours and fewer resources during off-hours [2, 9, 256]. For example, two CNFs might need one CPU core each during peak hours. During off-hours, the two CNFs might comfortably fit on one core. The TPC can thus be more than halved by placing the two CNFs on one core during off-hours.

The TPC KPIs thereby is at odds with the latency and throughput KPIs. Optimizing the TPC can increase the risk of core-tipping in the presence of unexpected traffic increases.

D2A is successful from an operational perspective if the resulting assignment strategies have higher throughput, lower latency, and lower TPC compared to baseline algorithms.

5.2.3 Formal problem description

Fig. 5.5 shows that an assignment of CNFs to cores respecting the CFS' operation can improve the KPIs throughput and latency. Specifically, if the platform co-locates CNFs on cores such that the assignment does not result in core-tipping, then all CNFs can process the packets arriving during one scheduling period. This maximizes throughput and minimizes latency. Previous work advocating the assignment of multiple CNFs to one core does not optimize towards avoiding core-tipping [190, 262]. D2A fills this gap using a data-driven approach to avoid core-tips and learns algorithms for two popular NP-complete optimization objectives: BP and LB [16, 264].

BP thereby aims to minimize the TPC, i.e., the goal is to use as few cores as possible while ensuring that no core tips. In contrast, LB aims at spreading the computational demand of CNFs across the available cores. An operator might choose a LB policy over a BP policy to reduce the risk of core-tipping as much as possible.

Formally, the BP objective of minimizing the number of used cores can be expressed as [264]:

$$\min \sum_{m \in \mathcal{M}} y_m, \tag{5.16}$$

being subject to Constraints (5.17)-(5.20). Constraint (5.17) states that the load of CNFs j must not exceed the capacity of core m:

$$\sum_{j \in \mathcal{J}} \mathrm{l}(j, m)\, x_{m,j} \leq \mathrm{c}(m), \quad \forall m \in \mathcal{M}, \tag{5.17}$$

where $\mathcal{J}$ is the set of all CNFs, $\mathcal{M}$ the set of cores, $x_{m,j}$ indicates if CNF j is assigned to core m, and the functions $\mathrm{l} : \mathcal{J} \times \mathcal{M} \rightarrow \mathbb{R}_{+}$, and $\mathrm{c} : \mathcal{M} \rightarrow \mathbb{R}+$ return the load CNF j induces on core m, and the capacity of core m in cycles. Constraint (5.17) is a simple approximation for the complex nonlinear interactions of CNFs resulting in core-tips.

Constraint (5.18) ensures that each CNF j is assigned to one and only one core m:

$$\sum_{m \in \mathcal{M}} x_{m,j} = 1, \quad \forall j \in \mathcal{J}. \tag{5.18}$$

Constraint (5.19) states that variable y_m is one if at least one CNF is assigned to core m:

$$y_m = \begin{cases} 1 & \text{if } \sum_{j \in \mathcal{J}} x_{m,j} > 1 \\ 0 & \text{else} \end{cases} \qquad \forall m \in \mathcal{M}. \tag{5.19}$$

Constraint (5.20) states that the variables $x_{m,j}$ are binary:

$$x_{m,j} \in \{0, 1\}, \quad \forall m, j \in \mathcal{M} \times \mathcal{J}, \tag{5.20}$$

The LB objective can be formulated as minimizing the maximum load on cores [16]:

$$\min \left(\max_{m \in \mathcal{M}} \mathrm{l}(m) \right), \tag{5.21}$$

subject to the Constraints (5.17), (5.18), and (5.20).

The BP and LB problems are a good fit for a learned heuristic that can exploit the problem-specific structure and include system effects [28, 285, 308]. As Sec. 5.6 shows, the simplification of core-tipping in Constraint (5.17) limits the model and results in decreased performance in practice. Note that simplification is common in the formulation of capacitated problems [11, 18, 135]. D2A overcomes the simplification with NCO and a data-driven DT. The DT predicts if a core tips, i.e., replaces the linear model in Constraint (5.17) with a learned one. NCO learns algorithms using the predictions of the DT, allowing the algorithms to avoid core-tips

5.3 Requirements engineering and high-level design

This section describes the design goals of D2A in Sec. 5.3.1, explains the components of D2A in Sec. 5.3.2, and the workload generation in Sec. 5.3.3. This section contains aspects of MaLANe's *Translate Objective* and *Generate Data* activities.

5.3.1 Operational Requirements

D2A is a SFC platform that allows CPU sharing between CNFs. D2A uses an NCO-based algorithm to assign CNFs to cores. In principle, D2A can continuously adapt during operation. For example, new SFCs and CNFs, or a change in the processing cost or arrival rate, can trigger a re-assignment. In a cluster, a separate orchestration tool is responsible for placing CNFs on individual D2A instances. This orchestration tool can rely on D2A components to make admission control decisions, e.g., use the learned heuristics and the DT to check whether a D2A instance could handle the assigned CNFs.

D2A uses built-in tools for process management in Linux. Hence, it does not require changes to the underlying operating system. To achieve an efficient CPU assignment, D2A has four key properties:

Local operation. D2A uses the data of the node it is running on to make assignment decisions. Thus, D2A does not require a complex distributed control plane during operation.

Generalization. LB and BP are combinatorial optimization problems with an exponentially growing solution space [16, 264]. D2A should learn strategies that generalize to perturbations of the problem instances. For example, the traffic might follow diurnal patterns [171], and the number and types of CNFs, as well as the SFCs deployed to a host, can vary. D2A should work for the resulting changed COP instances without re-training. For example, D2A should generalize to a varying number of different CNFs that could be deployed together in a specific scenario.

Non-preemptive operation. During operation, new SFCs or CNFs are expected to arrive over time. An SFC platform should find good assignments for newly arrived instances without necessarily changing the assignment of previously deployed CNFs.

Incremental deployment and fallback. Incremental deployment of D2A in production systems must be possible to ensure correct operation. In the case of performance degradation, D2A should allow for switching back to a previous working version or a basic algorithm like Round Robin.

The requirements put constraints on the system architecture, the training process, specifically the training data generation, and the ML models.

5.3.2 D2A system design

Fig. 5.7 shows the components of D2A : NF platform, Orchestrator, Monitor, Assigner, NFManager and School for Assigner. The Orchestrator coordinates the start of D2A . The Orchestrator receives a list of NFs chained to SFCs with estimated packet processing costs of the CNFs and packet arrival rates for each SFC. If this information is unavailable before deployment, then D2A can start with a default placement, estimate those numbers, and re-assign the CNFs. The Orchestrator passes this information to the Assigner. The Assigner responds with an assignment of CNFs to CPU cores. The Orchestrator then starts the NFManager of `OpenNetVM` and the CNFs. The NFManager also adjusts the weight parameter for the `cgroups` of CNFs based on Eq. (5.8) to implement RC [262].

The Monitor is integrated into the NFManager of `OpenNetVM` and observes the deployed CNFs, the cost of and traffic to CNFs, and how CNFs are chained. The School uses DTs and information from the Monitor to generate synthetic workloads and mimic the real system's behavior to teach Assigners. Once an

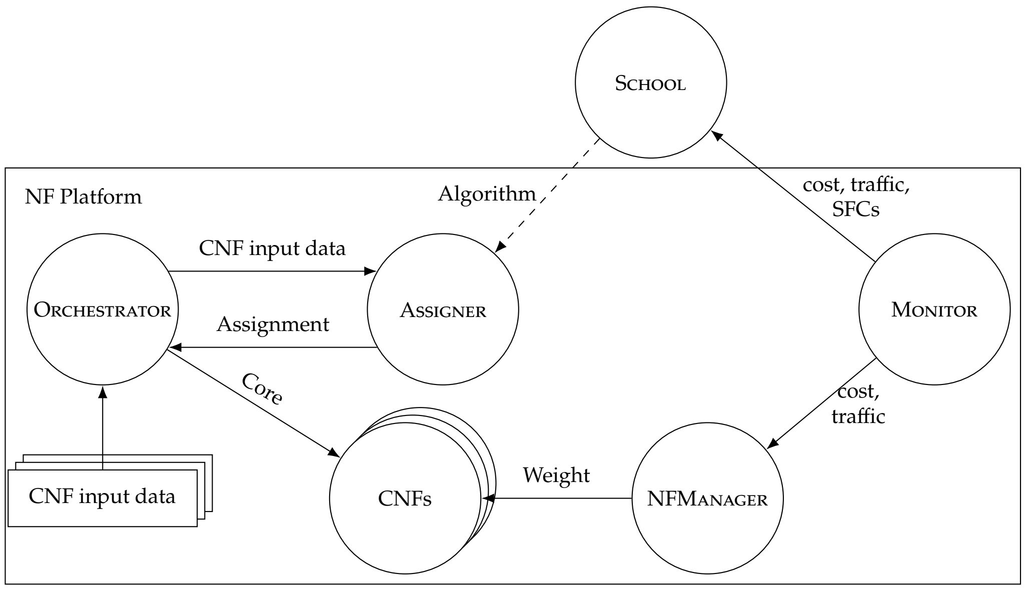


Figure 5.7 Overview of the pipeline.

Property	Values
#SFCs	$\{1, 2, \ldots, 8\}$
#CNFs per SFC	$\{1, 2, \ldots, 8\}$
Total #CNFs	$\{4, \ldots, 32\}$
#CPUs	16
#Dummy Loops	$\{0, 1, \ldots, 130\}$
Arrival rate of SFCs	$[0.02\,\text{Mpps}, 2.5\,\text{Mpps}]$
Concentration parameter α	5

Table 5.1 Properties of generated problem instances.

ASSIGNER graduates, it can be deployed to NF platforms and replace the previous ASSIGNER, e.g., a default algorithm such as Round Robin or a previously learned ASSIGNER. The NF platforms can benefit from improved assignments and revert to a basic algorithm should the current ASSIGNER be incompetent.

The assignment of CNFs happens once at startup or during operation. Further, events such as horizontal and vertical scaling decisions can trigger a re-assignment. Similarly, the ORCHESTRATOR can decide to re-assign if measurements from the MONITOR deviate from the ones the ORCHESTRATOR assumes.

As the evaluation in Sec. 5.6 shows, teaching ASSIGNERS with a simple underlying system model is enough to learn assignments superior to existing strategies for a specific scenario. Sec. 5.6 further shows that DTs learned from monitored data improve the system performance further. Moreover, Sec. 5.6 shows that the learned algorithms generalize to previously unseen problem instances.

Algorithm 3: Problem Generation Algorithm

```
   Input: maxNumSfcs $\in \mathbb{N}$, numCpus $\in \mathbb{N}$, maxMembers $\in \mathbb{N}$, $\alpha \in \mathbb{R}$, cpp
          $\in \mathbb{N}^M$, $\lambda_r$
   Output: sfcs, sfcRates
1  cpuAvailable $\leftarrow$ 1;
2  sfcAvailable $\leftarrow$ 1;
3  sampledVnfs $\leftarrow$ 0;
4  numSfcs $\sim$ U($\{1, \dots,$ maxNumSfcs$\}$);
5  sfcs[numSfcs];
6  cpus[numCpus];
7  sfcRates[numSfcs] $\sim \lambda_r \cdot$ Mul (Dir ($\alpha$ ones (numSfcs)));
8  for $i \in \{1, \dots, numSfcs\}$ do
9      sfcs[i]$\leftarrow$ {};
10 for $i \in \{1, \dots, numCpus\}$ do
11     cpus[i]$\leftarrow$ {};
12 while cpuAvailable and sfcAvailable and sampledVnfs < maxNumVnfs do
13     $j \leftarrow$ idxOfRandomAvailableSfc(sfcs);
14     compute $\leftarrow$ U $\left(\{0, \dots, \mid \text{cpp} \mid\}\right)$;
15     cpu $\leftarrow \emptyset$;
16     while cpu= $\emptyset$ and compute > 0 do
17         cpu $\leftarrow$ findNextFreeCpu(cpus, compute, sfcRates[$j$]);
18         if cpu = $\emptyset$ then
19             compute $\leftarrow compute - 1$;
20     if cpu = $\emptyset$ then
21         cpuAvailable $\leftarrow$ 0;
22     else
23         sfcs[$j$] $\cup$\{cpp[compute]\};
24         cpu $\cup$\{cpp[compute] $\cdot$ sfcRates[$j$]\};
25         sampledVnfs $\leftarrow$ sampledVnfs + 1;
26     if $\forall i \in \{1, \dots numSfcs\}$ :| sfcs[$i$] |> maxSfcs then
27         sfcAvailable $\leftarrow$ 0;
```

5.3.3 Problem Workload Generation

NCO can learn algorithms through RL that operate NF platforms at unmatched efficiency by exploiting patterns in the problem instances [259]. The intuition of using NCO is that the underlying RL algorithms produce heuristics with superior performance in *specific* scenarios, i.e., the typical workload of *one* operator. This idea aligns with the current trend of data-driven algorithm design, where algorithms are learned for a specific distribution over problem instances. The learned algorithms generalize to problem instances from that distribution and outperform existing heuristics [28, 285, 308].

To make the problem challenging, the remainder of this section designs an algorithm that produces a distribution over problem instances by varying the number of CNFs, SFCs, available CPU cores, processing costs, and arrival rates. The generated problems do not contain problem-specific patterns that RL could exploit and is, thus, a challenging benchmark. Tbl. 5.1 shows the dimensions of the generated data, and Fig. 5.11 shows one exemplary instance.

Problem generation algorithm Algorithm 3 shows the pseudo-code for the problem generation. The algorithm takes as input a maximum number of SFCs `maxNumSfcs`, several CPU cores `numCpus`, a maximum number of CNFs per SFC `maxMembers`, a scalar value α, and an array `cpp` that stores computational costs for CNFs.

The algorithm generates a problem instance as follows. First, Line 4 randomly samples the number of SFCs between one and a maximum number `maxNumSfcs`. Line 7 generates the arrival rate for individual SFCs by sampling a Multinomial distribution from a Dirichlet distribution with concentration parameter α, and multiplies the Multinomial with the system arrival rate λ_r. The concentration parameter controls the probability mass of the resulting Multinomial distribution.

Then, the algorithm iteratively constructs SFCs as follows. First, the algorithm randomly samples an SFC that has less than the maximum number `maxMembers` CNFs in Line 13. In Line 14, the algorithm then samples a random index into the `cpp` array.

Next, the algorithm attempts to find a core that can support the CNF in Line 17. The function findNextFreeCpu uses a first-fit approach, i.e., it iterates over the cores and returns the first core with enough remaining capacity to serve the demand, i.e., the product of packet arrival rate and computational cost. If no such CNF exists, the algorithm decrements the index into the `cpp` array in Line 19. The algorithm repeats these steps until it finds a core or the `compute` variable is less than zero.

If the algorithm finds a free core, the algorithm adds the CNF to the SFC by noting the computational cost in Line 23. Similarly, the algorithm accounts for the expected demand of the CNF on the CPU in Line 24. Then, the algorithm checks if there are still SFCs with space for additional CNFs in Line 26.

If the algorithm does not find a free core, the algorithm concludes that no more free cores are available and sets the variable `cpuAvailable` to zero in Line 21.

Overall, the algorithm iterates these steps until the problem either has a maximum number of CNFs, all SFCs have a maximum length or all cores are fully utilized. Then the algorithm terminates and returns the constructed SFCs, together with their rates. The generated problem instances should be feasible but still challenging. This allows a fair comparison between assignment algorithms, i.e., heuristics should find a solution to the problem.

Obtaining computational cost and rates Since an extensive library of microservice-based VNFs does not yet exist and implementing those is not the purpose of this work, the evaluation follows the same approach of using dummy CNFs as done in previous work [262, 305]. The CNFs for evaluating `D2A` are based on the `SimpleForward` example from `OpenNetVM`. The `SimpleForward` looks up the next hop of the current packet and updates corresponding metadata in `OpenNetVM`. The packet processing cost of the `SimpleForward` is varied by executing a `for`-loop with a configurable number of iterations. This results in fine-grained control over the computational cost of CNFs, allowing the generation of a wide range of workloads.

To obtain the processing costs in the `cpp` input, i.e., CPU cycles per packet, of CNFs, the modified `SimpleForward` is executed. The average processing cost for a varying number of loops in $\{10i \mid i = 0, 1, \ldots, 130\}$ is measured. The costs range from 80 to 8 027 cycles. Consequently, the CNFs can process between 25.5 Mpps and 0.275 Mpps.

The system arrival rate λ_r is upper-bounded to 2.5 Mpps, since measurements showed that `OpenNetVM` could reliably serve this rate. A packet arrival rate of 2.5 Mpps is enough in many scenarios, e.g., backbone links could be accommodated with this volume [309].

5.4 Learning Platform Design

This section introduces the heart of `D2A` : the learning platform. Sec. 5.4.1 and Sec. 5.4.2 explain the role of Game Theory and RL in `D2A` . Sec. 5.4.3 introduces the NA and Sec. 5.4.4 the used RL training algorithm. Thus, this section reflects MaLANe's *Formalize Problem* and *Prepare Training* activities.

5.4.1 Game Formulation

The assignment problem can be interpreted as a job scheduling game [283] and cast to a sequential game of perfect information. The game consists of a finite set of selfish players $\mathcal{J}$, which correspond to CNFs. Each player's action, i.e., strategy space, corresponds to the available CPU cores $\mathcal{M}$.

The players choose their actions one after the other, i.e., in sequence. All chosen actions, and the players themselves (how many, properties such as packet arrival rate or computational cost), are common knowledge. After every player has chosen her action, each player j receives a payoff $\pi(j) \in [-10, 0)$. The player's payoff later serves as a reward for RL. The payoff for the LB objective is:

$$\pi_{LB}(j) := \begin{cases} -10 & \text{if Constraint (5.17) is violated,} \\ -\max_{m \in \mathcal{M}} \left(\frac{\sum_{k \in \mathcal{J}} \mathrm{l}(k,m)}{\mathrm{c}(m)} \right) & \text{else.} \end{cases} \tag{5.22}$$

The payoff for the BP objective is:

$$\pi_{BP}(j) := \begin{cases} -10 & \text{if Constraint (5.17) is violated,} \\ -\sum_{m \in \mathcal{M}} \frac{\mathrm{l}(j,m)}{\sum_{k \in \mathcal{J}} x_{k,m} \mathrm{l}(k,m)} & \text{else.} \end{cases} \tag{5.23}$$

π_{LB} and π_{BP} are -10 if player j violates a constraint. Else, the player gets the maximum load across all cores (π_{LB}), or the load ratio, i.e., the fraction of the job's load on the core's total load (π_{BP}). π_{LB} is shared, i.e., all players that do not violate a constraint get the same payoff. A payoff of -10 arises during the game, i.e., only players whose action violates a constraint get a payoff of -10. The payoff of the other players is calculated at the end of the game.

Both games are finite and sequential and have at least one SNE in pure strategies [282]. Further, the Price of Stability (PS) is one [283], i.e., the ratio of highest social welfare[10] across all SPNEs of the game, and the maximum social welfare is one. Thus, the assignment of CNFs to cores with which maximum throughput, i.e., the optimal solution to the underlying optimization problem, is achieved is an SNE of the game. If the optimal assignment would not be an SNE of the game, then one player could change her strategy and receive a higher payoff. This is possible only if the throughput of the player is currently restricted, i.e., the core the player is on is overloaded. Thus, the target core the player changes to must have a smaller load than the current core the player is located on. This, in turn, would contradict the assumption that the system is in the optimum state.

Formulating the assignment problem as a sequential game has two advantages: It makes `D2A` flexible and interpretable. Flexible because players can have varying objective functions. Players violating a constraint can be sued without wrongly blaming others. The sequential nature of the game allows `D2A` to predict placements for arbitrary subsets of CNFs. For example, an operator can include co-location constraints of CNFs by incorporating these constraints into the utility functions. Thus, `D2A` would learn to co-locate corresponding CNFs, and other CNFs could adapt their strategies to this constraint. Interpretable because the

[10]Social welfare is the sum of individual player's payoff. In this case, the sum of the throughput of all CNFs [282].

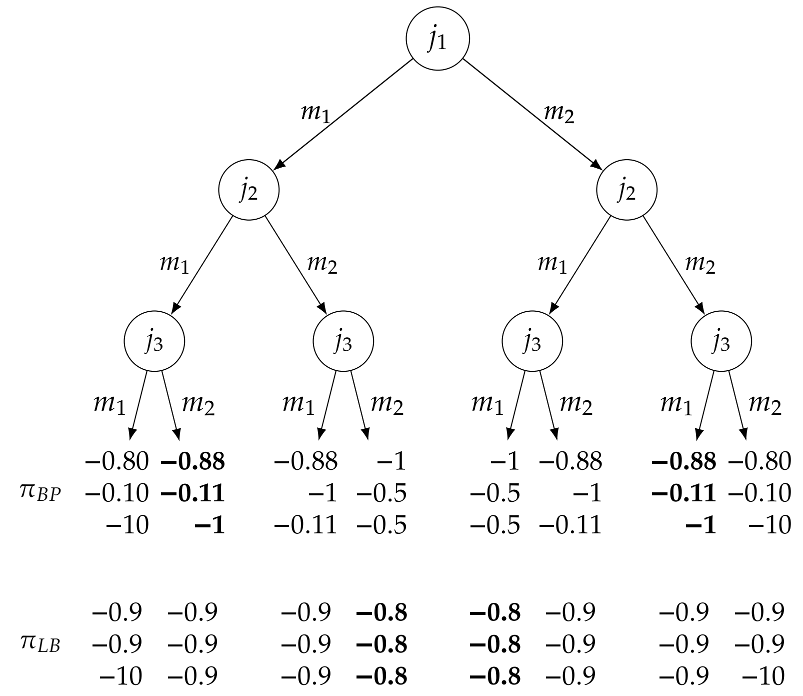

j_1
m_1 m_2
j_2 j_2
m_1 m_2 m_1 m_2
j_3 j_3 j_3 j_3
m_1 m_2 m_1 m_2 m_1 m_2 m_1 m_2
-0.80 $\mathbf{-0.88}$ -0.88 -1 -1 -0.88 $\mathbf{-0.88}$ -0.80
π_{BP} -0.10 $\mathbf{-0.11}$ -1 -0.5 -0.5 -1 $\mathbf{-0.11}$ -0.10
-10 $\mathbf{-1}$ -0.11 -0.5 -0.5 -0.11 $\mathbf{-1}$ -10
-0.9 -0.9 -0.9 $\mathbf{-0.8}$ $\mathbf{-0.8}$ -0.9 -0.9 -0.9
π_{LB} -0.9 -0.9 -0.9 $\mathbf{-0.8}$ $\mathbf{-0.8}$ -0.9 -0.9 -0.9
-10 -0.9 -0.9 $\mathbf{-0.8}$ $\mathbf{-0.8}$ -0.9 -0.9 -10

Figure 5.8 This figure shows an exemplary game in extensive form. The game has three players $\mathcal{J} = \{j_1, j_2, j_3\}$ and two cores $\mathcal{M} = \{m_1, m_2\}$. The utilities of the players for π_{BP} and π_{LB} are given at the bottom. Bold numbers indicate the Backwards Induction outcome.

game itself can be analyzed, allowing operators to conjecture on the potential behavior of the system.

For example, consider the extensive form game in Fig. 5.8. The game has three players $\mathcal{J} = \{j_1, j_2, j_3\}$ with two strategies each $\mathcal{M} = \{m_1, m_2\}$. Without loss of generality, let the capacity of the cores $\mathrm{c}(m_1) = \mathrm{c}(m_2) = 1$, and a load of player j_1 be $\mathrm{l}(j_1, m) = 0.8 \quad \forall m \in \mathcal{M}$, and for players, j_2 and j_3 be $\mathrm{l}(j_2, m) = \mathrm{l}(j_3, m) = 0.1 \quad \forall m \in \mathcal{M}$ on both cores. Fig. 5.8 shows the resulting utilities for each player for both utility functions π_{BP} and π_{LB} below the decision nodes of player j_3. Fig. 5.8 indicates the Backward Induction outcomes with bold numbers.

Fig. 5.8 shows that in the case of π_{BP}, the players maximize the load on the cores. In particular, player j_2 always co-locates with j_1, forcing player j_3 to choose the other core.

Similarly, in the case of π_{LB}, Fig. 5.8 shows how the players minimize the maximum load. In particular, players j_2 and j_3 always choose a different strategy from player j_1. Beyond avoiding the company of j_1, players j_2 and j_3 have no other objective. This becomes relevant if a third core m_3 is added to the game. The best utility value of -0.8 would remain unaffected, i.e., j_2 and j_3 have no incentive to choose separate cores. The evaluations in Sec. 5.6 will show this artifact.

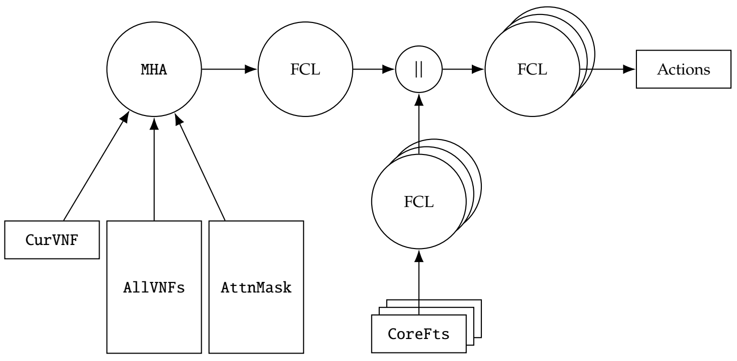


Figure 5.9 The architecture of the NN for payoff prediction.

5.4.2 The Role of RL

RL performs two tasks: (1) to predict each action's payoff for a player in a certain contingency, i.e., predicting the expected payoff for player *j* of choosing action *a* given the previously played actions, the player making its move, and the players that have yet to make their decision; (2) to incorporate the predictions of the DT, e.g., if a core will tip, into the learned algorithms, thus improving `D2A` 's performance.

Predicting the expected payoff can be interpreted as predicting the outcome of Backwards Induction [310]. Backward Induction requires the evaluation of every possible strategy profile, i.e., every possible assignment of CNFs to CPU cores [282]. Since this number grows exponentially, Backward Induction is not feasible in practice. Thus, `D2A` uses RL to learn a function parameterized through a NN that predicts the expected payoff. That is, `D2A` uses RL to learn the best responses, resulting in an SNE of the game [310]. RL thus learns the local optima of the underlying optimization problem guided by the game dynamics.

RL can include difficult-to-model system effects into the decision problem, because RL treats the game as a black-box. RL does not require the player's payoff functions to be linear, differentiable, or continuous [28, 296–298], which allows the learning from a DT of the system that itself uses ML to model system aspects, e.g., when a core is likely to tip. In principle, RL could even learn directly from the experience of the hardware platform [296–298].

In addition, RL automates the process of adjusting `D2A` to an operator's specific workload. The operator only has to adapt the distribution over problem instances for the algorithm training and can incorporate additional requirements such as co-location constraints. The operator can achieve both through a few lines of program code.

5.4.3 Neural Network Architecture

Fig. 5.9 shows the NN architecture used to predict the expected payoff for all actions of one CNF. The inputs are: `CurVNF`, `AllVNFs`, `AttnMask` and `CoreFts`,

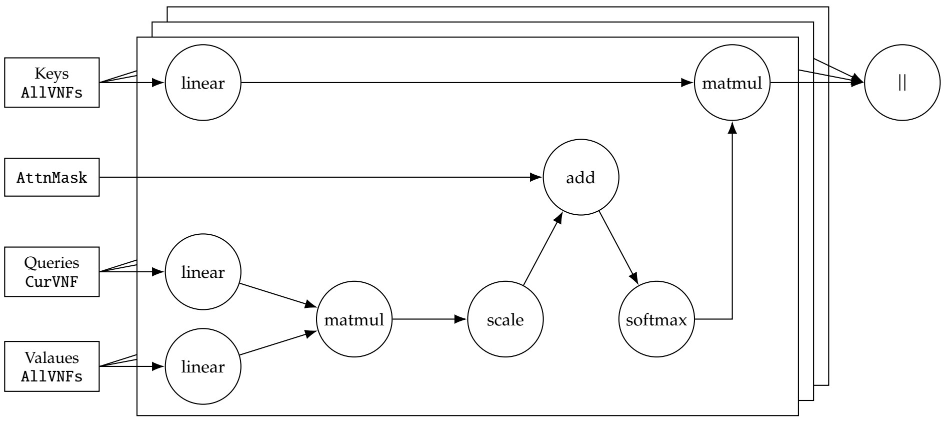


Figure 5.10 Processing in the Multi-Head Attention Module.

which are also the observation space for RL. The size of the output layer corresponds to the number of cores, which corresponds to the action space of RL.

`CoreFts` Features of CPU cores reflect the played actions. Each CPU core is represented by its relative load, the number of players that chose the core, the summed arrival rate, computational cost, and the minimum and maximum load ratio of players that chose this core. The relative load of a core m is:

$$\frac{1}{\mathrm{c}(m)} \sum_{j \in \mathcal{J}} x_{j,m} \mathrm{l}(j, m), \tag{5.24}$$

i.e., the sum of loads of CNFs that chose m divided by the cycles of m. The `CoreFts` is thus a vector in $\mathbb{R}^{|\mathcal{M}| \times 6}$. Each core's feature vector is transformed into a latent space through a sequence of FCLs. Each FCL transforms its input linearly and applies the Rectified Linear Unit (ReLU) activation function [311]:

$$\mathrm{ReLU} : \mathbb{R} \to \mathbb{R}_{>0}, x \mapsto \max(0, x). \tag{5.25}$$

At first glance, this representation has nothing to do with representing taken actions. The intuition underlying this representation is the question: Does the action sequence contain information that is strictly necessary to determine a best response? The answer is no. The sequence itself does not contain valuable information. The important information for the current player is to know which player is located on which core. Whether the corresponding player chose this core initially or right before the current player does not matter. Since the objective is to avoid core-tipping, the only information of interest is how the preceding agents affect the load level of the cores they have chosen.

Encoding the previously taken actions as a simple vector also simplifies the NN architecture drastically. Handling sequential data requires Recurrent Neural Network (RNN) or specially designed attention-based mechanisms. RNNs are difficult to train [284]. Attention cannot directly model sequences and thus requires additional experimentation with a positional encoding [185]. Both cases make training more challenging while not contributing toward better decision-making in this scenario.

CNF related input. Each CNF is represented with three numbers: The packet processing cost, the packet arrival rate in Mpps, and the demand, i.e., the product of arrival rate and cost. The processing cost is normalized to a value between zero and one. The load is divided by the clock speed, resulting in a value between zero and one. The resulting vectors are combined in the `AllVNFs` input, resulting in a $\mathbb{R}^{|\mathcal{J}| \times 3}$ matrix. The `AttnMask` is a tensor in $\{0, 1\}^{|\mathcal{J}| \times 1}$ and zeroes out players in `AllVNFs`. The `CurVNF` vector is one row from `AllVNFs` and corresponds to the CNF that currently makes its move. After the CNF made its move, the corresponding entry in `AttnMask` is set to zero, removing the CNF for the following players.

Relation between CNFs. The NN has a `MHA` module. This module helps the current CNF to set itself into relation to other CNFs [185, 312, 313]. Thus, the NN can learn which of the other players constitute important information for its action. The module consists of multiple attention mechanisms (called attention heads) using scaled-dot-product attention and a learnable non-linearity [185]. The output vectors from each attention head are concatenated to a single vector. `MHA` allows the NN to focus on different aspects of the input [185].

Fig. 5.10 shows the sequence of processing steps inside the `MHA` module. The attention mechanism takes as input three tensors: `Queries` Q, `Keys` K and `Values` V. The `Keys` and `Values` correspond to the `AllVNFs` tensor, i.e., are identical, and the `Queries` to the `CurVNF` vector. The inputs are linearly transformed through learnable weight matrices. The transformed `Queries` and `Keys` are used to calculate attention scores α as follows:

$$\alpha = \text{softmax}\left(\frac{QW^Q(KW^K)^T}{\sqrt{d_K}}\right) VW^V, \tag{5.26}$$

where W^Q, W^K and W^V are the linear transformations, i.e., learnable weight matrices, and d_K is the number of columns, of KW^K [185]. The attention mechanism then computes a convex combination of the `Values` V with the attention scores α. The `AttnMask` sets entries in α to zero, thus canceling rows in V. In essence, the `MHA` module learns to identify important entries in V given Q and K and combines them in its output. Each head in the `MHA` module can focus on different aspects, i.e., select different rows in V. Note that the attention mechanism is independent of the number of CNFs, i.e., rows in the `AllVNFs` tensor. Further, the attention mechanism is invariant to permutations of rows in the `AllVNFs` tensor [185]. The NN architecture can thus be used for a variable number of CNFs.

Action calculation. The `CoreFts` embedding and the `MHA` output are concatenated and passed through a sequence of FCLs, which finally produce the output, i.e., the expected payoff for the current CNF for any of the available CPU cores.

Thereby, the input data is exactly the data that is required for the game to have perfect information, i.e., the players know which contingency they are in and can act correspondingly [282]. `D2A` uses parameter sharing to reduce training time and sample complexity, i.e., uses the same weights for all CNFs. Individual CNFs are discriminated through the `CurVNF` input. Parameter sharing is common practice in Deep RL [312]. Further, the game formulation and NA allow the resulting NN to assign a single CNF, a subset of CNFs, or all CNFs to cores given the placement of the remaining CNFs. In an online scenario, new CNFs or SFCs can be integrated without touching the assignment of already running CNFs. Similarly, if changes in load or cost are detected, affected CNFs or SFCs can be re-assigned individually.

Algorithm 4: The algorithm takes as input sets of cores and CNFs, and returns a mapping from CNFs to cores.

```
  Input: $\mathcal{M}, \mathcal{J}$, NN
  Output: asgmt : $\mathcal{J} \to \mathcal{M}$
1 for $m \in \mathcal{M}$ do
2     asgmt$^{-1}(m) \leftarrow \emptyset$;
3 for $j \in \mathcal{J}$ do
4     asgmt$(j) \leftarrow \emptyset$;
5 sortDescending($\mathcal{J}$);
6 for $j \in \mathcal{J}$ do
7     $m \leftarrow$ getCore(NN, $\mathcal{M}, \mathcal{J}$, asgmt, asgmt$^{-1}$, $j$);
8     asgmt$(j) \leftarrow m$;
9     asgmt$^{-1}(m) \leftarrow$ asgmt$^{-1}(m) \cup \{j\}$;
```

RL-based assignment algorithm. Algorithm 4 illustrates the NN-based assignment algorithm executed from the Assigner entity. The algorithm takes as input a set of CPU cores $\mathcal{M}$, a set of CNFs $\mathcal{J}$, and a NN, and returns a mapping asgmt : $\mathcal{J} \to \mathcal{M}$ from CNFs to cores. Algorithm 4 first initializes asgmt in Line 4. Further, Algorithm 4 initializes an inverse assignment asgmt^{-1} : $\mathcal{M} \to 2^{\mathcal{J}}$ that returns the CNFs that are assigned to a core, if any. Then, Algorithm 4 sorts the CNFs based on their demand in descending order[11]. Then, the algorithm iterates over the CNFs. For each CNF $j \in \mathcal{J}$, Algorithm 4 invokes the NN in Line 7. The getCore function constructs the NNs inputs from $\mathcal{M}$, $\mathcal{J}$, asgmt, asgmt^{-1}, and the current CNF j, executes the NN that is passed to Algorithm 4, and returns a core $m \in \mathcal{M}$. The Algorithm 4 updates the two assignments in Line 8 and Line 9 and continues. After Algorithm 4 iterated over all CNFs, the algorithm ends and returns the assignment asgmt.

[11]This strategy is taken from the First Fit Decreasing (`FFD`) baseline algorithm. Sorting the CNFs improved the convergence of RL in experiments.

Note that Algorithm 4 does not check the capacity of cores. The algorithm takes the return value of the getCore function without further processing it.

5.4.4 RL training algorithm

The NN is trained with double Q-learning [314] with prioritized experience replay [315] and a dueling architecture [316] implemented in `RLLIB` [317].

Q-learning is a temporal difference method based on dynamic programming. In Q-learning, an agent learns action values of states: The expected cumulative reward the agent can obtain by taking action a in state s. Here, the action values, i.e., rewards, correspond to the payoff of the game, i.e., π_{LB} and π_{BP}. Here, the action-value function is approximated through a NN.

Double Q-learning uses a primary and a target network. The primary network is used for action evaluation, i.e., predicting the action values, and the target network is used for action-selection [314]. The primary network is trained on the regression loss between observed and predicted action values. The target network is updated using Polyak averaging. Double Q-learning helps to overcome variance in the reward function, e.g., if the best policy is associated with low reward during the first steps of an episode.

The replay memory improves sample efficiency and breaks temporal correlations between samples of one episode. Replay memories store transitions the learner generates during training. When updating the NNs, mini-batches of transition tuples are sampled from the replay memory. Samples with a high prediction loss are sampled with higher probability. By shuffling the samples in the memory, the temporal dependency is removed [315].

The dueling architecture is another technique to improve learning with RL. The neural network does not directly predict the action values in the dueling architecture. Instead, the dueling architecture predicts the state value and action advantages [316]. Intuitively, the dueling architecture can learn which states are valuable and which states are not without having to learn the effect of each action in each state. That is, the value function represents the expected value that the learner can hope to achieve once the learner is in that state. If the state value is small, then the action values are not that important. The learner should anyways try to avoid this state. The converse is true for states with high values. And then there are those states for which the actions have a major impact, i.e., an action can lead to states with low or high rewards. The dueling architecture decouples the identification of those situations.

Hyperparameters are tuned with the `ASHA` [238] and Population-Based Training (PBT) [318] implemented in `Tune` [239].

Symbol	Definition	Description
$\mathcal{J}$		Set of CNFs that are assigned.
$\mathcal{M}$		Set of available CPU cores.
$l(\cdot,\cdot)$	$\mathcal{J} \times \mathcal{M} \to \mathbb{R}$	Load a CNF puts on a CPU core.
$c(\cdot)$	$\mathcal{J} \cup \mathcal{M} \to \mathbb{R}$	Cost in cycles to process one packet for a CNF, or the cycles of a CPU core.
$r(\cdot)$	$\mathcal{J} \to \mathbb{R}$	Packet arrival rate for a CNF.
$x_{i,m}$		One if CNF i is assigned to core m, else zero.
$t(\cdot,\cdot)$	$\mathcal{J} \times \mathcal{M} \to \mathbb{R}$	Time slice of a CNF on a CPU core.
s_j	$\mathbb{R}$	Estimated rate.
$\hat{d}$		Maximum degree across problem graphs in the data set.
$\mathcal{G}$		Problem graph.
$\mathcal{E}$		Edges of a problem graph.
$\mathcal{V}$		Nodes of a problem graph.
$\mathcal{T}$		Set of CNF types.
τ	$\mathcal{V} \to \mathcal{T}$	Maps a node to a CNF type.
ft	$\mathcal{V} \to \mathbb{R}^n$	Maps a node to a real valued feature vector. The dimension of the feature vector depends on the type of node.

Table 5.2 Symbols used in Sec. 5.5.

5.5 Digital Twins

Obtaining training samples on the real system is expensive. Evaluating one assignment takes tens of seconds. A DT that mimics the system is thus an important contribution. The DT serves as an environment that RL uses to train algorithms. Sec. 5.5.1 introduces an analytical model of the system, and Sec. 5.5.2 introduces a method to predict the average CNF throughput and core-tipping from the system configuration.

5.5.1 Ideal System Model

The Ideal System Model (ISM) uses the packet arrival rate and computational cost of each CNF. The estimation is based on the `CFS` extended with `NFVNice`'s `RC` mechanism.

The model estimates the load that CNF j puts on core m:

$$l(j,m) = \mathrm{c}\,(j) \cdot r(j), \tag{5.27}$$

where $\mathrm{c}\,(j)$ returns the cost to process one packet in cycles, and $r(j)$ is the rate of CNF j in packets per second.

With the load, the model can estimate the time slice $t(j,m)$ granted to CNF j on core m:

$$t(j,m) = \Delta t_l \cdot \frac{1(j,m)}{\sum_{i\in\mathcal{J}} x_{i,m}\, 1(i,m)}, \tag{5.28}$$

where $\mathcal{J}$ is the set of all CNFs that are scheduled, Δt_l is the scheduling latency, and $x_{j,m}$ is a binary variable that is one if CNF j is assigned to core m and else zero. The number of packets processed by j during this time is:

$$s_j = \min\left(\gamma \frac{t(j,m)\,\mathrm{c}(m)}{\mathrm{c}(j)}; r(j)\right), \tag{5.29}$$

$\mathrm{c}(m)$ returns the clock speed of core m in cycles per second, and $\gamma \in (0,1)$ is a scalar factor to account for lost time due to context switches. The model can return an estimate of the throughput for the vanilla CFS by setting the cost and rate to some constant value: $c(j) = r(j) = \text{const.} \quad \forall j \in \mathcal{J}$.

The ISM compares the expected load on a core m against its capacity to predict core-tipping:

$$\sum_{j\in\mathcal{J}} l(j,m) < \mathrm{c}(m). \tag{5.30}$$

The model is simple and fast to evaluate. The model's correctness depends on the actual processing cost per packet. The processing cost can be affected through the interference of CNFs on, e.g., the last-level cache [18, 20]. The model can inject known interferences through the definition of $l(\cdot,\cdot)$, The model does not capture the complex behavior of the CFS, i.e., does not account for how the `vruntime` increases.

5.5.2 Learning-based Digital Twin

This section presents a concept that estimates KPIs such as per CNF throughput and core-tipping, i.e., violation of Constraint (5.17) from the system configuration. In a real deployment, the operator can learn a DT for his scenario with data gathered during operation or benchmarks.

Requirements

`OpenNetVM` and NF platforms are complex systems with many configuration options. For example, in an edge cloud scenario, the platform might be deployed in a small data center in which SFCs are deployed to multiple servers. The number of available CPU cores can be different on each server. In addition, the CNFs can interfere with each other in various ways [18–20, 287]. The DT must incorporate these properties to make correct predictions. Important aspects are:

- The number of TX threads.
- The number of available CPU cores.
- Chaining of CNFs to SFCs.
- Varying number of CNFs and SFCs
- Assignment of CNFs to CPUs.

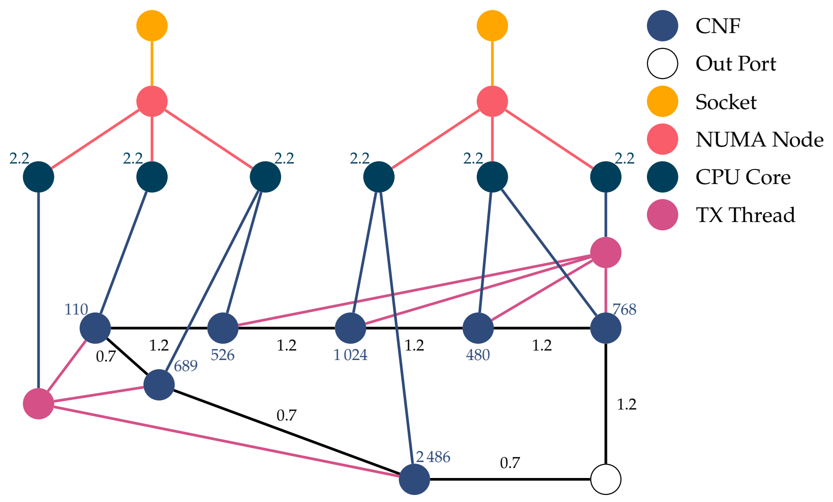


Figure 5.11 Exemplary problem graph. Numbers above CPU Core nodes are frequency in GHz, above CNF nodes are Cpp, and below edges between CNF nodes is the arrival rate in Mpps.

- Assignment of CNFs to TX threads.
- Co-location of CPUs in NUMA nodes.
- The output port.

`OpenNetVM` can configure the number of TX threads that copy packets between CNFs [269]. Depending on the length of the SFCs, the number of required TX threads can vary. Changes in the number of TX threads change the number of available cores since each TX thread blocks one core. Furthermore, servers can have a different number of CPU cores. The chaining of CNFs is important. Packet drops early in a SFC reduce the arrival rate of later CNFs. Furthermore, a dependency exists between TX threads, CPU cores, and CNF assignment. TX threads move data between the output and input buffers of CNFs [269]. Copying data to other CPU cores, CPU cores on other NUMA nodes, or even sockets increases the copying cost [20]. Further, the assignment of CNFs to cores and TX threads can vary. Lastly, the configuration of the system impacts the performance. Configuration options are: the NUMA node layout, the number of CPU sockets, and the forwarding of packets to the outgoing NIC [18, 287].

To condition the KPI prediction on those aspects, the data representation must include them. However, the complex dependencies make the representation of the data difficult.

Problem Representation

Each problem instance is represented as an un-directed, attributed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \tau, \mathrm{ft})$. Each node $u \in \mathcal{V}$ has one type $\tau(u) \in \mathcal{T}$ out of six types: vnf, tx, core, numa, socket and nic. The edges $\mathcal{E}$ describe the system's dependencies. The function $\mathrm{ft}\colon \mathcal{V} \to \mathbb{R}^n$ maps a node to a vector of node attributes:

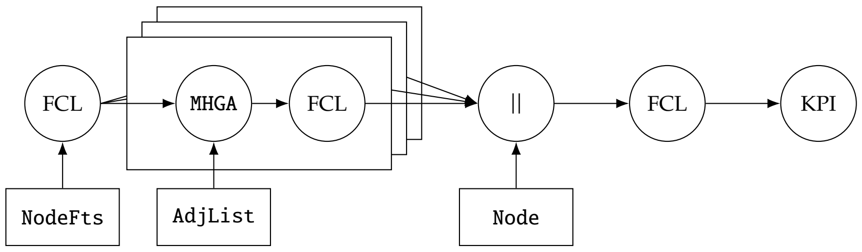


Figure 5.12 Neural Network of the learned twin.

- vnf: positional encoding, arrival rate, cost, demand, and load ratio.
- core: the sum of arrival rates, costs, and demands, and the ratio of minimum load ratio and maximum load ratio of adjacent vnf nodes.
- tx: the sum of incident vnf nodes' arrival rates.
- socket, numa, nic: have no additional attributes.

The positional encoding of vnf nodes represents their position in their SFC. The encoding is a vector in $\mathbb{R}^{10}$ and based on the encoding in the Transformer architecture [185].

Fig. 5.11 illustrates the representation. Fig. 5.11 shows seven CNFs that are chained to an SFC with two branches, two CPU sockets with six cores, three NUMA nodes, two TX threads and one outgoing NIC. Edges between core and vnf nodes represent the CNF to CPU assignment. Edges between tx and vnf nodes represent the CNF to TX thread assignment. Edges between vnf and nic nodes describe the chaining of CNFs to SFCs and the packet flow. Edges between core and numa nodes represent the organization of cores in NUMA nodes. Edges between socket and numa nodes indicate the socket a NUMA node is located on. Edges between core and tx nodes indicate the cores TX threads are running on.

The graph representation has four advantages: 1) The representation supports a variable number of CNFs, SFCs, TX threads, CPU cores, sockets, NUMA nodes and NICs; 2) The graph models complex interactions as edges; 3) nodes can be attributed with additional meta-data, or more node types can be included; 4) Learning approaches for graphs exist [319].

As the results in Sec. 5.6 show, the representation as graph indeed allows the prediction of different KPIs from the same input without further feature engineering.

Neural Architecture

A GATNN [266] model predicts KPIs from the graph representation. GATNNs use attention instead of RNNs or linear layers to aggregate node features. attention is the current state of the art in many areas of ML [185, 313]. GATNNs are a good choice for three reasons: 1) NNs and in extension GATNNs are general function approximators; 2) GATNNs can learn high-level features from low-level data; 3)

GATNNs can work on graphs with a varying number of nodes and edges, i.e., apply to problem graphs of varying size [266].

Using general function approximators is important since KPIs might not depend linearly on the input. For example, the CNFs' throughput does not depend linearly on the input, as Eq. (5.29) indicates, and the evaluation in Sec. 5.6 confirms. In addition, the complex dependencies between entities in the graph make feature engineering challenging. It is unclear how to aggregate the information in the problem graph into fixed-width feature vectors. Moreover, interactions and effects might change for different application scenarios. Engineered features working well in one scenario might not apply in another. NNs can help by learning aggregations and automatically engineering high-level features from low-level features, tuning them towards the prediction task [266].

Inputs and outputs Fig. 5.12 shows the neural architecture. The output is a KPI that the NN should predict. The output is calculated from four inputs that are derived from a problem graph $\mathcal{G}$: `NodeFts`, `AdjList`, `AttnMask`, and `Node`. `NodeFts` is a real valued tensor $\mathbb{R}^{|\hat{\mathcal{V}}| \times 27}$. Each node u in the problem graph $\mathcal{G}$ corresponds to one row in `NodeFts`. Each node u is represented with a one-hot encoding of its type $\tau(u)$ and its feature vector ft(u). ft(u) is inserted at a position based on the node's type. The remaining entries are zero.

The `AdjList` contains the neighbors for each node in the graph $\mathcal{G}$. `AdjList` is a tensor with natural numbers in $\mathbb{N}^{|\mathcal{E}| \times 2}$. Similarly, the input `AttnMask` is a tensor in $\{0, 1\}^{|\mathcal{V}| \times \hat{d} \times 1}$, where $\hat{d}$ is the maximum node degree across all graphs in the data set. `AttnMask[i, j, 0]` indicates if there is an edge between node i and its neighbor corresponding to j in `AdjList`. The NN uses `AttnMask` to handle nodes with a degree smaller $\hat{d}$. The values $\hat{d}$ and $| \hat{\mathcal{V}} |$ are needed to facilitate the training with mini-batches. The neural architecture can handle any number of nodes and node degrees.

The `Node` is a natural number, indexes into `NodeFts`, and identifies a node in $\mathcal{G}$ for which the NN should predict a KPI.

How the GATNN works. First, the NN in Fig. 5.12 embeds `NodeFts` into a latent space. Then, a sequence of Graph Attention Modules (`GAMs`) calculates new embeddings for each node based on its neighborhood. The outputs of the `GAMs` are concatenated, which improves the NN training [320, 321]. Then, the NN selects the CNF's embedding for which the NN should predict a KPI, and passes the embedding through a sequence of FCLs that produce the final output.

The GAMs. The `GAM` consists of a Multi-Head Graph Attention (`MHGA`) layer, and a FCL. The `MHGA` uses the `AdjList` and `AttnMask` to aggregate the embeddings of a node's neighbors. The layer uses a `MHA` mechanism with the node's embedding as `Queries`, and the embeddings of the node's neighbors and its own as `Keys` and `Values`. The FCL transforms the output of the `MHGA` layer into a latent space with the same dimension as `NodeFts` and allows the NN to learn non-linear dependencies.

For example, consider the initial CNF in Fig. 5.11. The `MHGA` module takes the feature vectors of the CNF itself, the neighboring CNFs, the TX Thread, and the CPU core and aggregates them into a new feature vector for the CNF. The new feature vectors thus contain information about the one-hop neighborhood of the node. In the second `GAM`, the process is repeated with the newly computed feature vectors. The CNF can thus learn about the neighbors of its neighbors. With each subsequent `GAM`, the feature vectors of the nodes are enriched with new information from other parts of the graph. This process happens in parallel for all nodes.

For training, a custom Graph Attention (GAT) layer is used. The original implementation of the GAT layer has a complexity of $\mathcal{O}(\mid \mathcal{V} \mid^2)$ since it computes the attention scores between all pairs of nodes and uses the adjacency matrix to mask out all non-neighbors [266]. Sparse matrix representations reduce complexity but do not allow the batching of multiple graphs [266], which is crucial to achieving high sample throughput in this use case. The custom implementation reduces the complexity to $\mathcal{O}(\hat{d} \mid \mathcal{V} \mid)$ by using the graphs' adjacency lists to gather the feature vectors of a node's neighbors and compute the attention scores directly over the neighborhood. That is, instead of the adjacency matrix, the custom GAT layer uses the adjacency list as input.

Limitations. The DT is limited by the data representation, ML models, and the training data. If the representation does not include necessary variables, or the ML models cannot represent the underlying system dynamics, then the predictions of the DT will be inaccurate. Further, the DT can only model interference present in the training data, which requires a deep understanding of the system and the collection of a comprehensive training set.

5.6 Evaluation

This section presents the evaluation results. Sec. 5.6.1 introduces baseline models for the DT. Sec. 5.6.2 introduces baseline algorithms to compare the learned assignment algorithms against. Sec. 5.6.3 discusses hyper-parameters for RL and the NN. Sec. 5.6.4 discusses settings for the SFC platform itself. Sec. 5.6.5 evaluates the quality of DTs, 5.6.6 compares learned algorithms against baselines, and Sec. 5.6.7 concludes this section. Thus, this section reflects MaLANe's *Train Model*, *Integrate Model*, *Investigate Data*, and *Deploy ML-enabled system* activities.

Parameter	Value
train batch size	379
buffer size	760 000
Attention heads	3
Attention dimension	26
Attention Dense	20
CPU Embedding	$\{102, 75, 94, 8\}$
Fully Connected	$\{52\}$
Exploration	EpsilonGreedy
initial epsilon	1.0
final epsilon	0.05
epsilon timesteps	531 901
learning rate	Annealed with population-based training.

Table 5.3 Hyper Parameter for model and training.

Parameter	Value
`CFS` Scheduling Latency	1 ms
`CFS` Scheduling Granularity	0.1 ms
Kernel tick rate `Hz`	250 Hz
Number of TX threads	4
Number of RX threads	1
`D2A` assignable CPUs	Cores $\{8, \ldots, 11\}$ on Socket 1, Cores $\{12, \ldots, 23\}$ on Socket 2
Packet Size	64 Byte
Arrival Rate	2.5 Mpps

Table 5.4 Parameter Settings of the `D2A` server.

Parameter	Value
#`GraphAttention-Modules`	6
#Attention heads	3
Initial Transform	20
Dim transform of `Keys`, `Queries`	10
Dim transform of `Values`	10
Attention Dense	20
Fully Connected	$\{92, 51\}$
learning rate	Annealed with PBT.
batch size	Annealed with PBT.

Table 5.5 Hyper Parameter for the learned digital twin.

5.6.1 DT Baseline

To show that the GATNN architecture proposed in Sec. 5.5.2 is competitive in the learning task, this section introduces two ML models that rely on domain knowledge and hand-crafted features for the prediction. The first model uses Logistic Regression to predict core-tipping. The second model uses Linear Regression to predict throughput. The remainder of this section introduces the models and explains the engineered features.

Logistic Regression

Logistic Regression is a basic ML model for binary classification [27]. Here, Logistic Regression should predict if a CPU core will tip given the assigned CNFs. Formally, Logistic Regression models the target variable as a probability:

$$\begin{aligned} \mathrm{p}_{\mathrm{logreg}} :& \{0,1\} \times \mathbb{R}^8 \times \mathbb{R}^8 \to (0,1), \\ & (y, x, \theta) \mapsto y \frac{1}{1-\exp(-\theta^T x)} + (1-y)\left(1 - \frac{1}{1-\exp(-\theta^T x)}\right). \end{aligned} \tag{5.31}$$

The binary variable y is the target variable and is 1 to represent a core-tip and 0 else. The variable $x \in \mathbb{R}^8$ is a feature vector characterizing a CPU core. The individual features are explained shortly. The variable $\theta \in \mathbb{R}^8$ are trainable parameters estimated from data using the graph representation of a problem instance defined in Sec. 5.5.2.

Let the node $u \in \mathcal{V}$ be a CPU core with type $\tau(u)$ = core, and $\mathcal{V}_u \doteq \{v \mid v \in \mathcal{V} \wedge (u,v) \in \mathcal{E} \wedge \tau(v) = \text{vnf}\}$ the set of CNFs assigned to core u. Then, the probability of a core-tip $\mathrm{p}_{\mathrm{logreg}}(1 \mid x, \theta)$ for one CPU core, is calculated from the following features:

1. Number of CNFs assigned to the CPU core:
$$\mid \mathcal{V}_v \mid . \tag{5.32}$$

2. The sum of the arrival rates of all CNFs assigned to the node:
$$\sum_{v \in \mathcal{V}_v} r(v). \tag{5.33}$$

3. The sum of the packet processing cost all CNFs assigned to the node:
$$\sum_{v \in \mathcal{V}_v} c(v). \tag{5.34}$$

4. The total demand, i.e., the sum of the demand of all CNFs assigned to the node:
$$\sum_{v \in \mathcal{V}_v} r(v)c(v). \tag{5.35}$$

5. The minimum demand ratio, i.e., the minimum CNFs demand relative to the total demand:
$$\mathrm{l}_{\min}(u) \doteq \frac{\min_{v \in \mathcal{V}_v} (r(v)c(v))}{\sum_{v \in \mathcal{V}_v} r(v)c(v)}. \tag{5.36}$$

6. The maximum demand ratio, i.e., the maximum CNFs demand relative to the total demand:
$$\mathrm{l}_{\max}(u) \doteq \frac{\max_{v \in \mathcal{V}_v} (r(v)c(v))}{\sum_{v \in \mathcal{V}_v} r(v)c(v)}. \tag{5.37}$$

7. The ratio of minimum and maximum demand ratio:
$$1 - \frac{\mathrm{l}_{\min}}{\mathrm{l}_{\max}}. \tag{5.38}$$

The last element in the feature vector is statically set to 1, and the corresponding entry in the parameters θ corresponds to an intercept.

The features defined in Eq. (5.32) to Eq. (5.38) can be interpreted as custom aggregation functions over the CNF nodes adjacent to a CPU node. The features reflect the CFS' behavior from Sec. 5.2.1. The system analysis showed that the total demand on the CPU core and a load of CNFs relative to each other is relevant.

Linear Regression

The evaluation uses linear regression to model the throughput a CNF achieves in the assignment. The estimation function is defined as follows:

$$\mathrm{linreg} : \mathbb{R}^{11} \times \mathbb{R}^{11} \to \mathbb{R}, (x, \theta) \mapsto \theta^T x. \tag{5.39}$$

Again, the argument $x \in \mathbb{R}^{11}$ corresponds to a feature vector for a CNF node derived from a problem's graph representation in Sec. 5.5.2. The parameter $\theta \in \mathbb{R}^{11}$ are trainable parameters. The feature representation of a CNF node $u \in \mathcal{V}$ with $\tau(u) = \mathrm{vnf}$ includes the feature representation of the CPU node u is assigned to, i.e., corresponds to the features defined in Eq. (5.32)-(5.38). Features eight to ten correspond to:

8. The CNF's arrival rate $r(u)$.

9. The CNF's computational cost $c(u)$.

10. The CNF's demand $r(u)c(u)$.

As before, the last element in the feature vector x is statically set to 1 for the intercept.

The intuition behind the linear regression is that the maximum throughput the CNF can achieve corresponds to its arrival rate. The arrival rate is part of the feature vector. The remaining features can then be mapped into an offset from the threshold that captures the expected deviation from the requested packet arrival rate.

5.6.2 Assignment baselines

The evaluation uses three baseline algorithms to evaluate the performance of the learned assignment algorithms: Round Robin (RR), Least Loaded First (LLF), and FFD. The remainder of this section explains the algorithms in detail.

Round Robin

Algorithm 5: Round Robin takes as input ordered sets of cores and CNFs, and returns a mapping from CNFs to cores.

```
  Input: $\mathcal{M}, \mathcal{J}$
  Output: asgmt : $\mathcal{J} \rightarrow \mathcal{M}$
1 for $i \in \{1, \ldots, |\mathcal{J}|\}$ do
2     asgmt($\mathcal{J}_i$) $\leftarrow \mathcal{M}_{i \bmod |\mathcal{M}|}$;
```

Algorithm 5 illustrates RR. Algorithm 5 returns a mapping from CNFs $\mathcal{J}$ to CPU cores $\mathcal{M}$ by iterating over the CNFs $\mathcal{J}$. Then, RR assigns each CNF to the jth core, where j is the modulo of the current CNF's loop index and the number of CPU cores.

Algorithm 5 shows that RR is static and does not consider previous assignment decisions. RR will use as many cores as possible, irrespective of the individual core's load.

Least Loaded First

Algorithm 6: Least Loaded First takes as input a set of cores and CNFs, and returns a mapping from CNFs to cores. LLF always chooses the core with the smallest load.

```
  Input: $\mathcal{M}, \mathcal{J}$
  Output: asgmt : $\mathcal{J} \rightarrow \mathcal{M}$
1 for $m \in \mathcal{M}$ do
2     $\text{asgmt}^{-1}(m) \leftarrow \emptyset$;
3 for $j \in \mathcal{J}$ do
4     $k \leftarrow \min_{m \in \mathcal{M}} \sum_{j \in \text{asgmt}^{-1}(m)} c(j) l(j)$;
5     asgmt($j$) $\leftarrow k$;
6     $\text{asgmt}^{-1}(k) \leftarrow \text{asgmt}^{-1}(k) \cup \{j\}$;
```

LLF is more advanced than RR and aims to keep cores evenly loaded. LLF is $2 - \frac{1}{|\mathcal{M}|}$ competitive [16].

Algorithm 6 illustrates the LLF algorithm. LLF takes as input the CNFs $\mathcal{J}$ and CPU cores $\mathcal{M}$ and returns an assignment asgmt : $\mathcal{J} \rightarrow \mathcal{M}$ from CNFs to cores. Algorithm 6 further tracks the CNFs that are assigned to a core with the mapping $\text{asgmt}^{-1} : \mathcal{M} \rightarrow 2^{\mathcal{J}}$. The mapping is initialized in Line 2.

Then, Algorithm 6 iterates over the CNFs in $\mathcal{J}$. Algorithm 6 retrieves the currently least-loaded core in Line 4. Then, Algorithm 6 updates asgmt and asgmt^{-1} in Line 5 and Line 6.

Similar to RR, LLF uses as many cores as possible. In contrast to RR, LLF takes previously made assignments into account.

First Fit Decreasing

Algorithm 7: Least Loaded First takes as input ordered sets of cores and CNFs, and returns a mapping from CNFs to cores.

```
   Input: $\mathcal{M}, \mathcal{J}$
   Output: asgmt : $\mathcal{J} \rightarrow \mathcal{M}$
1  for $m \in \mathcal{M}$ do
2      $\text{asgmt}^{-1}(m) \leftarrow \emptyset$;
3  for $j \in \mathcal{J}$ do
4      $\text{asgmt}(j) \leftarrow \emptyset$;
5  sortDescending($\mathcal{J}$);
6  for $j \in \mathcal{J}$ do
7      for $m \in \mathcal{M}$ do
8          if $c(j)r(j) + \sum_{w \in \text{asgmt}^{-1}(m)} c(w)r(w) < c(m)$ then
9              $\text{asgmt}(j) \leftarrow m$;
10             $\text{asgmt}^{-1}(m) \leftarrow \text{asgmt}^{-1}(m) \cup \{j\}$;
11             break;
12     if $\text{asgmt}(j) = \emptyset$ then
13         $m \leftarrow \text{LeastLoadedFirst}(\mathcal{M}, \{j\})(j)$;
14         $\text{asgmt}(j) \leftarrow m$;
15         $\text{asgmt}^{-1}(m) \leftarrow \text{asgmt}^{-1}(m) \cup \{j\}$;
```

FFD is a bin-packing algorithm. In contrast to RR and LLF, FFD's goal is to use as few CPU cores as possible. FFD has a competitive ratio of 1.7 [264].

Algorithm 7 illustrates the implementation of FFD used in the evaluation. Similar to Algorithm 6, Algorithm 7 uses an inverse mapping from cores to CNFs and initializes the mapping in Line 2. Further, Algorithm 7 initializes the assignment asgmt in Line 4. Then, Algorithm 7 sorts the CNFs in Line 5 based on their demand in descending order. Algorithm 7 calculates the demand for a CNF j as:

$$c(j)r(j) \tag{5.40}$$

Then, Algorithm 7 iterates for each CNF j over the cores $\mathcal{M}$. In Line 8, Algorithm 7 checks if the current core m has enough computational capacity left to accommodate the current CNF j. If m has enough capacity, Algorithm 7 updates the assignment and stops iterating over $\mathcal{M}$. If m has not enough capacity, the algorithm continues with the next core.

Once the loop over cores finishes, Algorithm 7 checks in Line 12 if a core has been found for CNF j. If asgmt returns the empty set, Algorithm 7 uses Algorithm 6 to find the currently least loaded core, and updates asgmt and asgmt^{-1} accordingly.

Algorithm 7 differs from the `FFD` algorithm in the literature by using `LLF` to find the least loaded core. If Algorithm 7 uses `LLF`, then no core has enough capacity left to accommodate the current CNF. Normally, `FFD` would open a new bin, i.e., CPU core in this case. However, the problem instance has only a finite set of cores, which Algorithm 7 has already exhausted. Since the CNF has to be assigned to some CPU core, Algorithm 7 uses `LLF` to minimize the amount by which a core is overloaded.

5.6.3 RL Hyperparameters

Tbl. 5.3 lists the hyperparameters used during learning. A batch size of 379, a replay buffer size of 760 000 samples, and the epsilon greedy exploration strategy is used. Initially, $\epsilon = 1$, i.e., the agents take fully random actions. The value of ϵ is annealed to $\epsilon = 0.05$ over 531 901 time steps. Once ϵ is annealed, five percent of the actions are random. The learning rate for updating the NN's parameters is not set to a fixed value. Instead, the learning rate is annealed with PBT throughout training with a population of five trials. During the evaluation, ϵ is set to zero, i.e., the agents greedily select the action with the highest predicted action value. All other hyperparameters are obtained with `ASHA`. `OpenAI`'s `Tune` and `Rllib` library are used to scale training [239, 317]. The NN resulting in the best average payoff for the players is then evaluated in the testbed on a separate set of problem instances.

The `MHA` module of the NN has three attention heads. Input tensors are transformed into a 26 dimensional latent space. The FCL following the `MHA` module has 20 neurons. The NN transforms the CPU features into an eight-dimensional latent space with a sequence of three FCLs with sizes $\{102, 75, 94\}$. The CPU feature embedding and the output of the `MHA` is concatenated and passed through one hidden layer with 52 neurons. The resulting network has 27 635 parameters and a size of 108 KBytes. The most expensive step is the calculation of the last hidden layer's activation, which requires $2(148 \cdot 52) + 52$ operations. The model is small, fits into a CPU cache, and assigns CNFs within milliseconds.

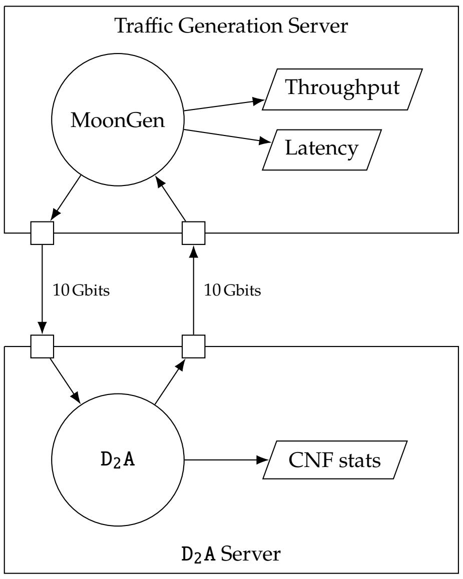


Figure 5.13 Testbed setup.

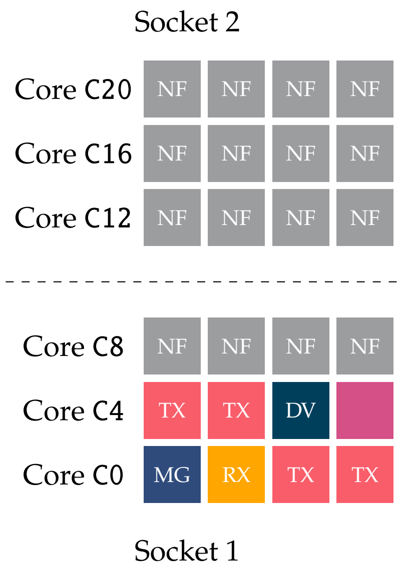


Figure 5.14 Core allocation.

5.6.4 Testbed configuration

Fig. 5.13 shows the testbed setup. The testbed has a traffic generation server and a server running D2A . The servers are connected back-to-back with two Intel 10G X550T NICs. Both servers have an Intel(R) Xeon(R) CPU E5-2650 with 24 cores each. The traffic generation server generates traffic with MoonGen [170], sends the traffic on one port to the D2A server, and receives traffic from the D2A server on the other port. MoonGen generates traffic for each SFC in a separate process and collects throughput and latency statistics. On the D2A server, D2A steers received traffic through the configured SFCs, sends the outgoing traffic back to the traffic generation server, and collects CNF statistics. To assign CNFs to cores, D2A uses the learned heuristics, and the three baseline algorithms RR, LLF, and FFD to evaluate 100 problem instances.

Tbl. 5.4 lists the hardware settings. The D2A server uses the CFS with a scheduling latency of 1 ms, a scheduling granularity of 0.1 ms, and a Hz value of 250 Hz. Fig. 5.14 shows the core allocation of D2A . The NFManager, a divider NF (DV), and TX and RX threads are allocated to cores C0 to C6. DV divides the incoming traffic for each SFC and forwards the traffic to the SFC's ingress NF. Core C7 is empty to leave room for OS processes running in the background. Cores C8-C23 are available to D2A for CNF assignment.

5.6.5 Digital Twin Performance

The ISM, and NN-based DT ($\mathtt{DT}_{\mathtt{NN}}$) are compared against Logistic Regression ($\mathtt{DT}_{\mathtt{LogR}}$) for detecting core-tipping and Linear Regression ($\mathtt{DT}_{\mathtt{LinR}}$) to predict throughput. $\mathtt{DT}_{\mathtt{NN}}$, $\mathtt{DT}_{\mathtt{LogR}}$, and $\mathtt{DT}_{\mathtt{LinR}}$ are trained on the measurements of 2 699 problems generated based on the procedure in Sec. 5.3.3. The problem's assignments are obtained with RR, LLF, FFD, $\mathtt{LB}_{\mathtt{HS}}$, and $\mathtt{BP}_{\mathtt{HS}}$. The resulting graphs have

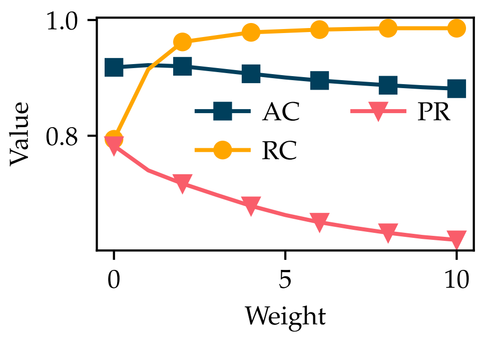


Figure 5.15 Accuracy (AC), recall (RC), and precision (PR) as a function of the class weight for $\mathtt{DT}_{\mathtt{LogR}}$.

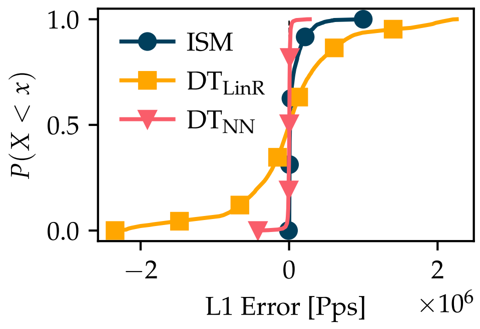


Figure 5.16 Deviation of ISM, NN, and $\mathtt{DT}_{\mathtt{LinR}}$ to the actually measured throughput.

between 38 and 68 nodes, 49 and 160 edges, and node degrees between 1 and 16. The data is split into a training, validation, and test set. The training set is used for parameter tuning, the validation set for hyperparameter tuning, and the test set to obtain the final results. The $\mathtt{DT}_{\mathtt{NN}}$ is trained with ASHA for initial hyper-parameter tuning and PBT to obtain a final model. The models for $\mathtt{DT}_{\mathtt{LinR}}$ and $\mathtt{DT}_{\mathtt{LogR}}$ were trained with the Scikit-Learn library [322]

DTs can detect core-tipping. The twins are evaluated based on accuracy, precision, and recall [323]. Here, a higher recall is better, i.e., if a core tips, the models will detect it. The $\mathtt{DT}_{\mathtt{NN}}$ achieves an accuracy of 93.40 %, recall of 77.18 %, and precision of 96.77 %. The $\mathtt{DT}_{\mathtt{LogR}}$ of 91.81 %, 79.38 %, and 78.27 %. The ISM of 80.63 %, 66.47 %, and 48.55 %. The $\mathtt{DT}_{\mathtt{NN}}$ has a high precision, i.e., few false positives. Precision and recall are similar for $\mathtt{DT}_{\mathtt{LogR}}$, which has the best recall score among the models. Still, a recall of 79.35 % means that 20 % of the core-tips will not be detected. Too much to reliably avoid core-tipping in learned assignments.

To improve the recall, the samples of tipped cores in the training set are given a higher weight in the objective function, i.e., the models are biased towards predicting this class. Fig. 5.15 shows how the metrics of $\mathtt{DT}_{\mathtt{LogR}}$ change. Fig. 5.15 shows that the recall increases, and the precision and accuracy decrease. A weight of 8 results in the highest recall of 98.58 %. Precision and accuracy decrease to 63.22 %, and 88.72 %. For the $\mathtt{DT}_{\mathtt{NN}}$, weighting the objective function had no strong impact on any metric.

	Avg. Throughput [%]				Avg. Total Cost per Packet [Cpp]				Avg. Latency [ms]			
	2.5 Mpps		1 Mpps		2.5 Mpps		1 Mpps		2.5 Mpps		1 Mpps	
	CFS	RC	CFS	RC	CFS	RC	CFS	RC	CFS	RC	CFS	RC
RR	93.95	96.51	99.71	99.71	2 084.23	2 035.19	4 981.98	4 982.12	1.64	1.80	**0.04**	**0.06**
LLF	96.14	96.98	99.71	99.71	2 041.64	2 005.59	4 982.20	4 982.14	1.19	2.45	0.07	0.06
$\texttt{LB}_{\texttt{HS}}$	96.94	98.64	**99.72**	99.71	1 745.45	1 713.73	3 296.24	3 296.24	1.19	1.06	0.06	0.07
$\texttt{LB}_{\texttt{Col}}$	98.03	98.83	99.68	99.71	1 861.44	1 837.51	4 670.68	4 668.99	**0.50**	0.82	0.05	0.06
FFD	67.19	77.53	77.07	83.57	**1 291.54**	**1 112.98**	**1 305.87**	**1 203.84**	7.07	8.36	5.71	7.43
$\texttt{BP}_{\texttt{HS}}$	88.08	88.21	98.60	98.18	1 343.98	1 330.69	1 635.19	1 636.04	4.80	4.57	0.84	0.82
$\texttt{BP}_{\texttt{Col}}$	**98.10**	**98.97**	99.71	**99.71**	1 587.16	1 572.24	1 791.62	1 791.64	0.81	**0.81**	0.15	0.21

Table 5.6 Latency, total cost per packet, and throughput for 2.5 Mpps and 1 Mpps.

$\texttt{DT}_{\texttt{NN}}$ predicts throughput accurately. Fig. 5.16 shows the target and prediction's differences as CDFs. Positive values indicate over-estimation and negative values under-estimation. Fig. 5.16 shows that the $\texttt{DT}_{\texttt{NN}}$ and the ISM have an error close to zero for more than 50 % of the samples. The ISM over-estimates the throughput most of the time, while the $\texttt{DT}_{\texttt{NN}}$ under-estimates and over-estimates. The $\texttt{DT}_{\texttt{NN}}$ and ISM have an average error of 2.53 % and 15.05 %. The $\texttt{DT}_{\texttt{LinR}}$ cannot predict the throughput and has an average error of 149.29 %.

In summary, the $\texttt{DT}_{\texttt{NN}}$ and $\texttt{DT}_{\texttt{LogR}}$ can detect core-tipping better than the ISM. By weighing the objective function, the $\texttt{DT}_{\texttt{LogR}}$ achieves a recall of > 98 %. The $\texttt{DT}_{\texttt{NN}}$ has a high precision of > 96 %. Thus, the remainder of the evaluation uses the $\texttt{DT}_{\texttt{LogR}}$ to train new generations of assignment algorithms. Further, the NN can predict the throughput of CNFs with an average error of 2.53 % of the actual throughput.

5.6.6 System Performance

The evaluation investigates the impact of RC over the vanilla CFS. MoonGen sends packets with 1 Mpps and 2.5 Mpps. The baseline algorithms RR, LLF, and FFD are compared against four learned assigners: $\texttt{BP}_{\texttt{HS}}$, $\texttt{BP}_{\texttt{Col}}$, $\texttt{LB}_{\texttt{HS}}$, and $\texttt{LB}_{\texttt{Col}}$. $\texttt{BP}_{\texttt{HS}}$ and $\texttt{BP}_{\texttt{Col}}$ use the π_{BP} utility function, and $\texttt{LB}_{\texttt{HS}}$, and $\texttt{LB}_{\texttt{Col}}$ the π_{LB} utility function. $\texttt{LB}_{\texttt{HS}}$ and $\texttt{BP}_{\texttt{HS}}$ graduated from High School (HS) and learned with the ISM. $\texttt{LB}_{\texttt{Col}}$ and $\texttt{BP}_{\texttt{Col}}$ graduated from College (Col) and learned with the $\texttt{DT}_{\texttt{LogR}}$. The assignment algorithms are compared based on throughput, TPC, and the end-to-end latency. The throughput and latency is measured with MoonGen. The TPC is measured on the D2A server, and is defined as:

$$T \frac{\sum_{m \in \mathcal{M}} y_m c(m)}{\sum_{j \in \mathcal{J}} \text{packets}(j)}, \tag{5.41}$$

where T is the experiment period, and packets(j) gives the total number of processed packets for CNF j. That is, TPC measures the cycles that the system spends on each packet. Each configuration of assignment algorithm, arrival rate, and scheduling policy is evaluated with 100 problems generated with the procedure in Sec. 5.3.3, resulting in a total of 8 400 experiments. Tbl. 5.6 gives an overview of all combinations. The remainder of this section details the result for an input rate of 2.5 Mpps, and RC. The results for other combinations are similar.

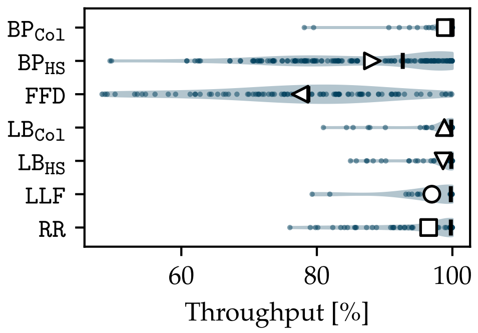


Figure 5.17 Violinplots of the throughput where markers indicate the average, vertical bars the median, and small dots actual samples.

D_2A achieves higher throughput. Fig. 5.17 shows a violinplot of the throughput. Markers represent the average (avg), black vertical bars the median throughput, and small dots are actual samples. Fig. 5.17 shows that BP_{Col} (avg. 98.97 %) is comparable to LB_{Col} (avg. 98.83 %), and outperforms LLF (avg 96.98 %) and RR (avg 96.51 %). FFD (avg. 77.53 %) and BP_{HS} (avg. 88.21 %) perform worse, which is expected. RR and LLF can use all 16 cores and co-locate at most two CNFs, which reduces the risk of core-tipping. FFD often tips cores, which degrades performance. BP_{HS} learns to avoid core-tipping through the ISM that detects 60 % of the cases. BP_{Col} further reduces core-tipping by learning with the DT_{LogR} model that detects 98.58 % of the cases.

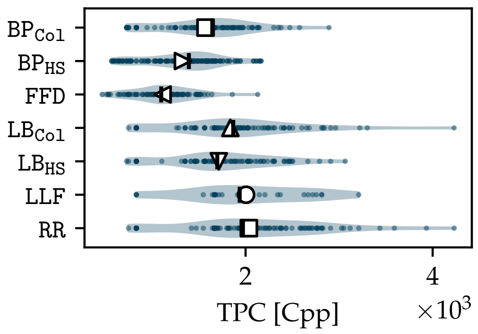


Figure 5.18 Violinplots of the TPC where marker indicate the average, vertical bars the median and small dots actual samples

D_2A uses fewer cycles per packet. Fig. 5.18 compares the TPC. RR and LLF have the highest TPC with more than 2 000 Cpp on average. The different shapes of the violines are caused by the dependence on the number of packets and used cores.

LLF and RR use equally many cores, but LLF has a higher throughput, resulting in more processed packets, which reduces the TPC.

LB_{HS} and LB_{Col} have a lower TPC with 1 713 Cpps, and 1 838 Cpps. LB_{HS} and LB_{Col} co-locate CNFs more than RR and LLF, which reduces the TPC. This behavior is a training artifact and not intended. The shared utility π_{LB} returns the maximum load. Thus, players can co-locate if the corresponding core does not become the new most loaded core.

The BP algorithms have the smallest TPC with 1 113 Cpps, 1 331 Cpps, and 1 572 Cpps for FFD, BP_{HS}, and BP_{Col}. This is expected, BP_{HS} and BP_{Col} use more cores to avoid core-tipping. BP_{Col} reduces the TPC compared to LB_{Col} by a factor of 1.17. For the 1 Mpps scenario, BP_{Col} reduces the TPC by a factor of 2.60.

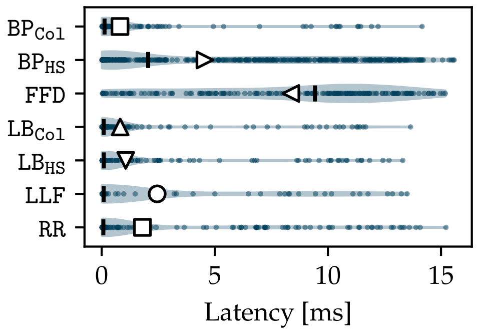


Figure 5.19 Violinplots of the latency where marker indicates the average, vertical bars the median and small dots actual samples.

D_2A reduces latency. Fig. 5.19 shows a violinplot of the latency. Fig. 5.19 shows that FFD has the highest latency with 8.36 ms on average, followed by BP_{HS} with 4.57 ms on average. RR, LLF and LB_{HS} is markedly smaller with average latencies between 1.0 ms and 2.5 ms. LB_{Col} and BP_{Col} achieve the lowest latencies with an average of 0.82 ms and 0.81 ms.

The heavy-tailed distributions of latencies in Fig. 5.19 reflect the behavior of the CFS described in Sec. 5.2. Specifically, the distribution of FFD shows that RC is insufficient to achieve low latencies and high throughput. Also, Fig. 5.19 shows that good assignments reduce latencies and jitter. For example, the distribution of BP_{Col} concentrates closer to the left, i.e., has smaller values than even the load balancing heuristics that use all cores in the system.

5.6.7 Discussion

This chapter shows that NCO can learn effective assignment strategies in varying scenarios and that the learned strategies generalize to previously unseen scenarios. Given the existing literature on SFC scheduling, the number of SFC and CNFs is representative. In fact, the scale of the problems in this chapter exceeds that of most related work. Still, the results are limited in that a uniform packet size and constant packet arrival rate for each SFC is assumed, and interference between CNFs mostly relates to the compute resources.

However, the arrival rates and individual SFCs arrival rates are random in the evaluation. Thus, the presented framework can be expected to handle changing arrival rates during continuous operation, although D2A has not been evaluated in such a setting. For example, the effect of the transient phase in which CNFs get re-assigned to different cores on throughput is unclear.

Further, there is reason to believe that CNFs will be more predictable in their behavior than monolithic VNFs. Since CNFs can be expected to do only a few specific tasks, the variance in processing cost will not be as strong as, e.g., reported for the Snort IDS [288].

In summary, there is reason to believe that in a real-world deployment, more and stronger patterns exist compared to the evaluation scenario. This strengthens the case for data-driven algorithm design in such scenarios. The evaluation scenarios had few patterns in the data that RL could exploit. Still, RL managed to improve upon baseline algorithms. Future work could thus include the implementation of an exemplary micro-service-based SFC to verify this hypothesis.

5.7 Summary

This chapter investigates NF platforms that allow CPU core sharing between CNFs. The chapter explains how the OS's scheduling policies cause interference between co-located CNFs and how the interference impacts throughput and latency. The chapter then presents D2A that learns a Load Balancing (LB) and a Bin-Packing (BP) heuristic with Neural Combinatorial Optimization (NCO) and Game Theory. The novel NCO and Game Theory combination make D2A incrementally deployable and easy to integrate into existing infrastructures. Extensive measurements in a testbed show that D2A increases throughput up to 46 %, and reduces latency by up to 93 %, with only a 23 % increase in computational resources compared to First Fit Decreasing. LB heuristics that achieve comparable throughput and latencies require 58 % more resources. Further, the chapter shows the potential of an entirely data-driven approach by integrating a Machine Learning (ML)-based Digital Twin (DT) of the NF platform into the training process of NCO. The DT models the complex interference of CNFs on the CPU and increases the learned algorithm's throughput by up to 11 %, and decreases latency by up to 90 % compared to using an analytical model.

This chapter opens interesting avenues for future work. In particular, more realistic CNFs and SFCs might increase the challenge of achieving efficient workload processing, which is a perfect application for D2A . This chapter proposes that these problems can benefit from the DT's ML model. Further, this chapter provides a rigorous analysis of the Linux scheduler's behavior in NF platforms that allow NF co-location, which can help improve those platforms. Lastly, the framework this chapter presents could be applied hierarchically. This chapter takes the set of CNFs as given. In a Cloud-Native Communication Network, the assignment of CNF to nodes must be made from another entity. This entity could use the same principle as D2A to learn assignment strategies on the node level and utilize the proposed DT structure to predict potential failures.

Chapter 6

Conclusion and Outlook

Technological trends such as the Internet of Things (IoT), CPSs, and the shift towards digital consumption drive network operators to adopt a cloud-native approach for networking using a MEC compute infrastructure. The resulting Cloud-Native Communication Network helps network operators to face increasing data traffic costs while meeting the vastly different requirements of applications such as XR, gaming, CPSs, and M2M while providing a distinguished service to succeed in the competition.

Network automation is one of the key benefits driving the adoption of Cloud-Native Communication Networks. However, application and infrastructure diversity make automation challenging. For example, algorithms automating one network function or service are not necessarily transferable to another. Even the design of control algorithms for one function or service can be challenging, as assumptions about its behavior might be too simple. Similarly, the hardware at different MEC locations might differ, reflecting control decisions for the function or service. Obtaining a sophisticated behavioral model of the function or service across different hardware stacks is a time-consuming and challenging task that cannot be realized at the scale of Cloud-Native Communication Networks. Further, automation requires control decisions that often correspond to solutions to algorithmic hard optimization problems.

The concept of data-driven algorithm design offers a solution to this challenge. Data-driven algorithms use ML and AI to automatically learn algorithms tailored to specific use cases from data. Further, data-driven algorithm design offers the opportunity to include complex behaviors and system effects into control algorithms that would otherwise be hard to realize. Data-driven algorithm design can be interpreted as an additional layer in the operation of networks that automates the design of algorithms automating network operation. Naturally, data-driven algorithm design introduces challenges and needs a structured approach to succeed. This thesis studies data-driven algorithm design in three application scenarios: Micro-service-based traffic classification, Traffic Engineering, and micro-service co-location on shared CPU cores, and summarizes the gained experience in a process model for data science projects in networking.

Section 6.1 summarizes this thesis's key contributions and outcomes. Sec. 6.2 gives an overview of interesting and challenging research directions in the area of data-driven algorithm design for network operation automation.

6.1 Summary

The main outcome of this thesis consists of a process model for performing data science projects in the networking domain, which summarizes the experience of over 30 data science projects of varying scopes. Further, this thesis presents frameworks to obtain data-driven algorithms for three application scenarios and shows the effectiveness of the resulting ML-enabled systems through implementations and measurements in testbeds.

A process model for data science projects in networking. The MaLANe process model abstracts and summarizes the experience of over 30 data science projects in the networking domain. MaLANe emphasizes the strong interdisciplinarily of data science in the networking domain and bridges gaps between disciplinary fields to improve collaboration. MaLANe makes the layers of data science projects in the networking domain explicit, i.e., links the business view, systems engineering, data science, and networking experts. MaLANe thereby complements existing process models for data science projects with a unique view on the prototypical realization of the resulting ML-enabled systems and a perspective on RL-based projects.

ProFi- Data-driven Websitefingerprinting. ProFi focuses on traffic classification, specifically, on WFP in an online deployment scenario and at scale. ProFi relies on a micro-service-based system design that makes it amenable to operate in the context of Cloud-Native Communication Networks. For example, ProFi could operate at edge clouds and provide visibility into accessed services in an encrypt-everything regime. To this end, ProFi relies on computationally cheap PGMs to capture a website's behavior on the flow level, enabling ProFi to operate live in the network. The PGMs further detects changes in the website's characteristics that require the refitting of the model. ProFi's modular architecture simplifies updating existing models and integrating new models. A comparison to SoA shows that ProFi achieves competitive performance wrt. classification metrics while outperforming SoA in inference speed and not assuming the availability of all traffic for one load of a website's webpage. Testbed evaluations show that ProFi can process more than 400 website accesses per second at 10 Gbit/s, and provides classification results within milliseconds.

MISTILL- Data-driven Traffic Engineering Mechanisms. MISTILL provides a method to learn TE mechanisms for TE policies in small and medium-sized Fat Tree topologies. Thus, MISTILL can improve TE at cloud locations by automatically providing TE mechanisms to TE policies that improve the performance for the specific applications accessed in each location. Specifically, MISTILL learns how to encode local information in switches into update messages, which information is required to predict the forwarding behavior for a given destination, and how to process the exchanged update messages to construct a path to the destination that follows the initial TE policy. MISTILL learns TE mechanisms for TE policies through an efficient training procedure that exploits the markovian structure of forwarding decisions. To operate in practice, MISTILL relies on the eBPF, pushes the route computation into the end-hosts, and uses source routing to enforce the forwarding decisions. MISTILL uses IP multicast to distribute the necessary update messages. Multicast enables the network to exchange only those updated messages necessary for the often sparse communication patterns. Source routing and multicast allow the deployment of MISTILL in legacy networks with COTS hardware. Packet-level simulations show that MISTILL learns TE mechanisms that closely implement a TE policy's forwarding decisions and generalize to unseen traffic patterns. Testbed evaluations show that MISTILL reacts within 1.3 ms to changes in the network.

D2A - Data-driven core sharing for CNFs. D2A facilitates the co-location of CNFs on one CPU core. D2A thus directly contributes to the efficiency of Cloud-Native Communication Networks. D2A uses ML to learn the interference of co-located CNFs and predicts if a core can no longer sustain the assigned load. Then, D2A uses the learned model together with RL to learn control algorithms implementing load balancing and bin packing heuristics. D2A introduces a novel framework that reformulates the classical optimization problem as congestion games and then uses RL to learn the best responses of the game. Testbed measurements comparing D2A to three popular load balancing and bin-packing heuristics show that D2A learns heuristics that avoid the overutilization of cores while also improving latency, processing cost and throughput of the ML-enabled system.

6.2 Future Work

The results of this thesis open many avenues for future work, of which the following ones are of particular interest.

Holistic and probabilistic approach to traffic classification. Traffic classification often relies solely on data directly captured from the observed traffic. However, often additional information is available. For example, the usage of applications and services is time-dependent and could depend on users' geo-location or correlate with previously used services and applications. PROFI's probabilistic framework allows the integration of this information by extending the underlying graphical models accordingly. Investigating how available information causally related to the observed traffic changes traffic classification performance is an interesting and challenging research question.

Mixture models with continuous emissions for traffic classification. The data analysis in this thesis shows that website behavior can be multi-modal and distributional concerning packet sizes. To capture these properties better, the PGMs could be adapted to a continuous emission model, i.e., directly modeling the packet size distribution at specific positions in the flow. Further, a mixture model could represent the different website behaviors. How to formulate, efficiently learn, and realize such a system is an interesting and challenging research question.

Learning TE mechanisms for arbitrary topologies. Fat Tree topologies are a popular DCN topology. However, other DCN topologies that are not based on a Clos topology and have a more irregular structure exist. Similarly, WAN topologies also have diverse and irregular topologies that are not based on a Clos topology. Extending the learning of TE mechanisms for TE policies to arbitrary topologies is an interesting and challenging research question.

Reinforcement Learning-based TE Mechanisms. Depending on the TE policy, generating the training data necessary to learn a TE mechanism in a supervised fashion might be computationally expensive specifically, if the TE policy should consider the temporal correlation between flowlets. Learning TE mechanisms with RL in such a setting would alleviate the training data generation. Similar to D2A , interferences in the network that are hard to model and include in traditional TE policies could be incorporated in the TE mechanisms. Formulating such TE policies and deriving a RL representation of the problem that allows the efficient learning of TE mechanisms is, therefore, an interesting and challenging research question.

NN offloading to NICs. Hosts in DCNs today are equipped with NICs that have strong computational capacities. The advanced computational capabilities of NICs could be used to improve the reaction time of Mistill. Specifically, the route computation could be moved directly onto the NIC, avoiding the communication overhead between user- and kernel space. Implementing a NN for the execution on NICs, further offloading the route computation, and measuring the effect on route computation is an interesting and challenging research question.

DT-based VNF to node scheduling. Cloud-Native Communication Networks often rely strongly on K8S to orchestrate the execution of containers on the infrastructure. K8S uses a scheduler to assign containers to available nodes. Using D$_2$A 's approach for learning assignment strategies could improve the overall performance further. For example, given D$_2$A 's learned assignment strategies and DTs, a second level could use these as building blocks to learn the assignment of containers to nodes such that nodes are not overloaded and latency, throughput, and computational effort are optimized. Integrating such a learned scheduler with container orchestration tools and measuring the resulting performance benefits is an interesting and challenging research question.

Appendix

Acronyms

D$_2$A Data-Driven Assignment 123, 124, 137, 141–143, 147, 148, 150, 153, 161, 167, 169–173, 177, 178, 184

AI Artificial Intelligence 2, 174

ALPN Application-Layer Protocol Negotiation 31, 34, *Glossary:* Application-Layer Protocol Negotiation (ALPN)

AR Augmented Reality *Glossary:* Augmented Reality

ASHA Asynchronous Successive Halving Algorithm 104, 107, 108, 154, 166, 168

ASIC Application-specific Integrated Circuit 87

ASUM-DM Analytics Solutions Unified Method for Data Mining/Predictive Analytics *Glossary:* Analytics Solutions Unified Method for Data Mining/Predictive Analytics

AVX Advanced Vector Instruction 2, 5, 113

BP Bin Packing 123, 141–143, 172

CDF Cumulative Distribution Function 62, 63, 140, 169

CDN Content Delivery Network 34, 35, 45, 60, 69

CFS Completly Fair Scheduler 125–130, 136, 138–141, 156, 163, 167, 169

`cgroup` control groups 128–130, 143, *Glossary:* Control Groups (`cgroups`)

CI Confidence Interval 68

CICD Continuous Integration Continuous Delivery 28, 29

CNF Containerized Network Function 1, 3, 7, 8, 23, 26, 122–125, 129, 130, 135–144, 146–148, 150–153, 155–160, 162–167, 169–173, 176, 183, *Glossary:* Containerized Network Function (CNF)

CNN Convolutional Neural Network 35, 84

COP Combinatorial Optimization Problem 2, 4, 14, 18–21, 25, 29, 135, 143, *Glossary:* Combinatorial Optimization Problem

COPTP Constraint Optimization Problem 88, 89, 92, 93, 98, 105, 182

COTS Commodity of the Shelf 70, 176

Cpp Cycles per packet 139, 140, *Glossary:* Cycles per Packet (Cpp)

CPS Cyber-physical System 1, 174

CPU Central Processing Unit 7, 8, 21, 23, 25, 29, 73, 74, 87, 113, 117, 118, 122, 123, 125–130, 136, 137, 139, 140, 142–144, 146, 147, 150, 151, 153, 155–158, 160–167, 172, 175, 176, 182

CRISP-DM CRoss Industry Standard for Data Mining 10–15, 17, 24–26, 29, 182, *Glossary:* CRoss Industry Standard for Data Mining

CRISP-ML(Q) CRoss Industry Standard Process model for the development of Machine Learning with Quality assurance methodology 14, 15, 29, *Glossary:* CRoss Industry Standard Process model for the development of Machine Learning with Quality assurance methodology

CSP Constraint Satisfaction Problem 88–90, 92, 93, 98, 104, 105

DCN Datacenter Network 3, 7, 80, 81, 83, 85–88, 92, 94, 95, 105, 113, 121, 177

DMME Data Mining Methodology for Engineering Applications 14, 29, *Glossary:* Data Mining Methodology for Engineering Applications

DNN Deep Neural Network 35

DNS Domain Name Service 31

DoH DNS over HTTPS 31

DoT DNS over TLS 31

DPDK Data Plane Development Kit 124, 126, 137, *Glossary:* Data Plane Development Kit (DPDK)

DPU Data Processing Unit 121

DST Data Science Trajectories 14, 15

DT Digital Twin 123, 124, 142, 143, 150, 155, 156, 160, 167, 168, 172, 173, 178, *Glossary:* Digital Twin (DT)

eBPF extended Berkeley Packet Filter 79, 112, 113, 115–118, 176, *Glossary:*

ECH Encrypted Client Hello 31

ECMP Equal Cost Multi-Pathing 89–94, 104, 106–109

ESNI Encrypted Server Name Indication 31, *Glossary:* Encrypted Server Name Indication (ESNI)

FCL Fully Connected Layer 107, 151, 153, 159, 160, 166

FFD First Fit Decreasing 153, 164–167, 169–171

FIB Forward Information Base 84

FMEA Failure Modes and Effects Analysis 14, *Glossary:* Failure Modes and Effects Analysis

FPGA Field Programmable Gate Array 2, 87

GAM Graph Attention Module 159, 160

GAT Graph Attention 160

GATNN Graph Attention Neural Network 124, 158, 159, 162

GNN Graph Neural Network 84, 85

GPU Graphical Processing Unit 2, 117–119

HMM Hidden Markov Model 36, 39–42, 52, 54, 182

HNSA Hidden Node State Advertisement 87, 98–101, 109, 111–116, 118–120, 183

HTTP Hypter Text Transfer Protocol 67

HTTPS Hypter Text Transfer Protocol Secure 31

ILP Integer Linear Program 81

INT In Network Telemetry 23, *Glossary:* Inband Network Telemetry (INT)

IoT Internet of Things 174, *Glossary:* Internet of Things

IP Internet Protocol 34, 44, 45, 58, 59, 64, 72, 99, 101, 104, 105, 113–115, 176

IPMC IP Multicasting 112–114, 116

ISM Ideal System Model 155, 156, 167–170

K8S Kubernetes 1, 70, 177, 178

KDD Knowledge Discovery in Databases 13, 14, *Glossary:* Knowledge Discovery in Databases (KDD)

KLD Kullback-Leibler Divergence 108

kNN k-Nearest Neighbor 51, 57, 58, 61, 62, 64, 65, 68, 69

KPI Key Performance Indicator 27, 28, 84, 85, 140, 141, 156–159

LB Load Balancing 123, 141–143, 172

LCP Least Cost Path 89–94, 105, 108, 112

LLF Least Loaded First 164–167, 169–171

LSA Link State Advertisement 98

M2M Machine-to-Machine 1, 174, *Glossary:*

MAC Medium Access Control 102

MaLANe Machine Learning Applications in Networking 9–11, 14–16, 19, 22, 24, 26, 28–30, 33, 34, 36, 44, 60, 70, 80, 85, 88, 106, 112, 124, 137, 142, 147, 160, 175

MAuC Message Authentication Code 38

MC Markov Chain 32, 39, 40, 48, 51, 52, 56, 58, 61–63, 65–69, 74, 182

MDP Markov Decision Process 134, 135

MEC Multi-access Edge Computing 1, 32, 45, 70, 76–78, 122, 174, *Glossary:* Multi-access Edge Computing (MEC)

MHA Multi-Head Attention 82, 100, 101, 104, 107, 108, 152

MHGA Multi-Head Graph Attention 159, 160

ML Machine Learning 2–5, 7–15, 17–24, 26–35, 48, 68, 70, 76, 78–80, 83–87, 105, 113, 121, 123, 124, 135, 136, 143, 150, 158, 160, 162, 172, 174–176, *Glossary:* Machine Learning

MLC Maximum Likelihood Classifier 48, 60

MLOps Machine Learning Operations 29, *Glossary:* Machine Learning Operations (MLOps)

MPLS Multiprotocol Label Switching 112, 113, 117

MTU Maximum Transmission Unit 38

NA Neural Architecture 7, 79, 80, 82, 86, 87, 97, 98, 100, 101, 120, 123, 136, 147, 153

NAT Network Address Translation 34

NCE Noise Contrastive Estimation 102, 103

NCO Neural Combinatorial Optimization 123, 124, 133, 135–137, 142, 146, 171, 172

NE Nash Equilibrium 130–133

NF Network Function 1, 122, 123, 125, 135–137, 146, 156, 172, 183, *Glossary:* Network Function

NFV Network Function Virtualization 70, 122, 124, *Glossary:* Network Function Virtualization (NFV)

NIC Network Interface Card 113, 116, 121, 125, 136, 157, 158, 167, 177

NLL Negative Log Likelihood 67, 68, 182

NLP Natural Language Processing 82

NN Neural Network 7, 26, 27, 79, 80, 83–87, 92, 95, 97–101, 104, 105, 107–109, 111–113, 116–121, 135, 150–154, 158–160, 166–169, 177, 183

NUMA Non-Uniform Memory Access 136, 157, 158

NV Network Virtualization *Glossary:* Network Virtualization

OPEX OPerational EXpenditures 17, *Glossary:* Operational Expenditures (OPEX)

OS Operating System 21, 123, 136, 137, 167, 172

OSPF Open Shortest Path First 83

OVS OpenVSwitch 113

P2P Peer-to-Peer 43

PBT Population-Based Training 154, 161, 166, 168

PCAP Packet Caputure File 73

PGM Probabilistic Graphical Model 32, 33, 48–51, 62, 63, 65, 67–72, 74, 75, 77, 175, 177, 182

PHMM Profile Hidden Markov Model (PHMM) 32, 42, 43, 48, 52–58, 61–63, 65–69, 73–75, 182

PoC Proof-of-Concept 26, 79, 80, 112–115, 117, 119–121

ProFi PRObabilistic website FIngerprinting 32–35, 38, 39, 42–53, 57, 64, 66, 68–71, 75–77, 175, 176, 182

QoE Quality of Experience 31

QoS Quality of Service 31

QUIC Quick UDP Internet Connections 36

RC rate-cost proportional fairness 130, 169, 170

ReLU Rectified Linear Unit 151

RL Reinforcement Learning 7, 10, 12, 20, 24–26, 29, 124, 133–136, 146–148, 150, 151, 153–155, 160, 172, 175–177, *Glossary:* Reinforcement Learning (RL)

RNN Recurrent Neural Network 151, 158

RR Round Robin 164, 165, 167, 169–171

RT Real Time 137, 139

RTLSRS Random TLS Record Size Defense 50, 64, 66, 69, 182

SEMMA Sample, Explore, Modify, Model, Access 13, 14, *Glossary:* Sample, Explore, Modify, Model, Access (SEMMA)

SFC Service Function Chain 1, 8, 122, 123, 125, 136, 137, 142–144, 146, 147, 153, 156–158, 160, 167, 171, 172, *Glossary:* Service Function Chain (SFC)

SGD Stochastic Gradient Descent 92, 95, 97

SI State Interest 98, 100, 101

SL Supervised Learning 10, 12, 20, 24, 25, *Glossary:* Supervised Learning (SL)

SLD Second Level Domain 39

SNE Subgame-perfect Nash Equilibrium 130, 131, 133, 148, 150

SNI Server Name Indication 31, 34, *Glossary:* Server Name Indication (SNI)

SNMP Simple Network Management Protocol 21

SoA State-of-the-Art 10, 18, 32, 45, 80, 124, 175

SP (Communication) Service Provider 1, 2, 31, 32, 76–78, *Glossary:* (Communication) Service Provider (CSP)

SR Segment Routing 112, 113, 120

SSH Secure Shell 43

SSL Secure Socket Layer 50, 69, 76

SVM Support Vector Machine 35, 51, 64, 65, 87

T2L Time to Label 75, 76

TCP Transmission Control Protocol 36, 38, 47, 60, 71–73, 80

TE Traffic Engineering 1–3, 7, 8, 18, 23, 27, 78–81, 83–89, 92, 94–101, 105–108, 113, 115, 117, 119–121, 176, 177, 183, *Glossary:* Traffic Engineering

TLS Transport Layer Security 7, 31–38, 43–47, 50, 52, 56, 61–64, 67–69, 71, 72, 76, 77, 184

ToR Top-of-the-Rack 81–83, 88, 90, 97, 103, 104, 112, 113, 115

Tor The onion router 31, 76

TPC Total Packet Cost 140, 141, 169–171

TPU Tensor Processing Unit 2

UDP User Datagram Protocol 70–73, 105, 115

UL Unsupervised Learning 10, 20, 24, 25, *Glossary:* Unsupervised Learning (UL)

URL Unified Resource Locator 39, 58

VM Virtual Machine *Glossary:* Virtual Machine

VN Virtual Network 1, *Glossary:* Virtual Network

VNF Virtual Network Function 1, 70, 71, 84, 122, 123, 125, 135–137, 172, *Glossary:* Virtual Network Function

VPN Virtual Private Network 34, 45

WAN Wide Area Network 64, 177

WCMP Weighted Cost Multi-Pathing 89, 90, 93–95, 98, 105, 108, 112

WFP Website Fingerprinting 31–36, 39, 43–45, 50, 57, 63, 64, 69, 70, 75–77, 175

XR eXtended Reality 1, 174, *Glossary:* eXtended Reality (XR)

Glossary

`OpenNetVM` A Network Function platform that uses DPDK to realize a zero-copy chaining of NFs on commodity hardware. 147

5G The fifth mobile network generation. 140

Application-Layer Protocol Negotiation (ALPN) An extension of the TLS protocol allowing the application to negotiate the protocol used during the connection. With the ALPN extension, the client can advertise, e.g., different HTTP variants that it supports, without adding additional round trip times. 31

cloud-native Implementation of applications based on micro-services, service meshes, immutable infrastructure, and containers. 122

Cloud-Native Communication Network Networks are implemented as containerized, micro-service-based applications ontop of a (distributed) cloud infrastructure. 1–3, 122–124, 129, 140, 173–177

Control Groups (`cgroups`) A Linux kernel tool allowing the management, quotation, and accouting of various ressources such as CPU, memory, disk I/O, network, etc. 128

CRoss Industry Standard for Data Mining A comprehensive process model for managing data mining projects. 10

CRoss Industry Standard Process model for the development of Machine Learning with Quality assu
A process model that builds on top of the CRISP-DM process model and includes Machine Learning specific aspects, and continuous quality assurance during the development process. 14

Cycles per Packet (Cpp) The (average) cost of a µVNF to process one packet. Expressed in CPU cycles. 139

Data Mining A process that extracts information from data. Data Mining starts from a relatively clear business goal. Necessary data is already collected and available for further processing. The existence of the wanted information in the data is established, i.e., it is known that the data contains the sought-for information. The result of Data Mining is a way to extract, i.e., mine the information from the data. 9, 10, 12–15

Data Mining Methodology for Engineering Applications Process model that extends the CRISP-DM process model with additional steps that are specific to data mining tasks in the industrial and engineering context. 14

Data Plane Development Kit (DPDK) An open-source software project managed by the Linux Foundation containing libraries enabling the packet processing in the user space. Specifically, DPDK moves the processing of packets from the kernel to the user space. Thereby, DPDK operates in poll-mode, i.e., consistenly fetches packets from the NIC. 124

Digital Twin (DT) A digital representation of a physical or immaterial object. Digital twins enable data exchange, and causal inference, i.e., predict how the twinned object would react to certain inputs. 123

Encrypted Server Name Indication (ESNI) TLS extension that encrypts the SNI to hide the actual service a user is accessing on a CDN. 31

eXtended Reality (XR) Umbrella term referring to computer-generated environments merging the physical and virtual worlds or creating fully virtual worlds. In contrast to AR, XR is not restricted to vision and can extend to smell, touch, hearing, and taste. 1

Failure Modes and Effects Analysis The FMEA is a method for risk management and assessments. The goal of the FMEA is the reduction of failures in production. 14

Inband Network Telemetry (INT) Allows the monitoring, collection, and reporting of fine-grained network statistics directly in the data plane. INT can be realized with special packets or using normal data packets. INT devices in the network add their information directly to the packets. No special control plane is required. 23

Internet of Things Interconnects physical and virtual objects and enables information exchange between them. 174

Knowledge Discovery in Databases (KDD) Process model for the analysis of structured and unstructured data. 13

Machine Learning A methodology that uses computational methods to automatically detect and describe patterns in data 2

Machine Learning Operations (MLOps) Application of established DevOps concepts to ML-enabled systems. MLOps improves the efficiency of developing, deploying, managing, and monitoring ML models. 29

Machine-to-Machine The automated communication exchange between non-human entities such as sensors, robots, vehicles, etc. 1

Multi-access Edge Computing (MEC) Provides (distributed) cloud-computing capabilities and an IT service environment at the network edge to application developers and content providers. MEC provides ultra-low latency, high bandwidth and real-time acecss to radio network information to applications. 1, 32

Network Function Specific task or functionality in a network that is traditionally implemented as a middlebox. Examples are firewalls or load balancers. 1

Network Function Virtualization (NFV) Uses virtualization technologies to virtualize network functions into software appliances running on commodity hardware. Common examples of NFs include load balances, firewalls, intrusion detection devices, and mobile network functions. 70

Operational Expenditures (OPEX) The cost necessary to operate a system, infrastructure, or business. 17

Reinforcement Learning (RL) An area of ML where an agent is situated in an environment with unknown dynamics. The agent's task is the optimization of a numerical reward signal through trial and error. During this process, the agent learns the environment dynamics, i.e., how to contol the environment such that the numerical reward signal is optimized. 7

Sample, Explore, Modify, Model, Access (SEMMA) A process model consisting of five stages developed by the SAS Institute. The SEMMA process model guides the implementation of data mining applications. The process model ist tailored towards a product of the SAS Institute. 13

Server Name Indication (SNI) SNI is a TLS extension that allows the access of multiple encrypted websites through a single server, even if this server is accessible via one IP address only. The SNI is included in the TLS `ClientHello` message in cleartext. 31

Service Function Chain (SFC) A SFC consists of an ordered set of VNFs and allows the creation of a composite service. An SFC applies to all, or a part of the network traffic. 1

Six Sigma Six Sigma is a scientific method for process improvement. Six Sigma relies heavily on statistics and defines a process with five steps to manage projects for process improvement. 14, 17

SmartNIC A SmartNIC is a special type of Network Interface Card (NIC) that allows the offloading of potentially compute intensive tasks from a server's

CPU. For example, encryption and packet segmentation can be offloaded to SmartNICs, freeing ressources on the server. Further, SmartNICs can be programmable, i.e., allow the execution of custom tasks directly on the NIC. 84, 87

Supervised Learning (SL) SL is an area of ML relying on labeled data, i.e., consisting of datapoints that have features and an associated label. The label can be categorical or numerical. For categorical data, the resulting ML application is called classification. For numerical data, the resulting ML application is called regression. 10

The extended Berkeley Packet Filter (eBPF) The eBPF can run sandboxed programs in a privileged context, e.g., in the kernel of an operating system. With eBPF, the kernel's capabilities can be extended at runtime without having to the kernel source code, or load kernel modules. In networking, the eBPF can be used to realize custom packet processing algorithms. 79

thread Object of activity within an executing program aka process. Traditionally, one executing program had one thread. Modern applications use multiple threads. Each thread has a unique program counter, process stack, and set of processor registers. Threads are the entities that are scheduled by the Linux scheduler. 126

Traffic Engineering Refers to the analysis, design, and optimization of data flow and routes data takes in communication networks. 1, 78

Unsupervised Learning (UL) A type of algorithm that extracts patterns from unlabeled data. Most commonly, UL is associated with clustering. Other common applications correspond to representation learning, e.g., learning embeddings for words in a text corpus or nodes in a graph. 10

Virtual Network A virtual network is a software-based administrative entity that contains hardware and software resources, as well as network functions. A virtual network runs on top of a physical network. 1

List of Figures

List of Tables

Bibliography

Publications by the author

Journal articles

[29] Patrick Krämer, Philip Diederich, Corinna Krämer, Rastin Pries, Wolfgang Kellerer, and Andreas Blenk. "D2A: Operating a Service Function Chain Platform with Data-Driven Scheduling Policies". In: *IEEE TNSM* (2022), pp. 1–15. DOI: `10.1109/TNSM.2022.3177694`.

[30] Patrick Krämer, Oliver Zeidler, Johannes Zerwas, Andreas Blenk, and Wolfgang Kellerer. "Mistill: Distilling Distributed Network Protocols from Examples". In: *IEEE TNSM* (2023), pp. 1–16. DOI: `10.1109/TNSM.2023.3263529`.

[31] Patrick Krämer et al. "ProFi: Scalable and efficient website fingerprinting". In: *IEEE TNSM* (2023), pp. 1–14. DOI: `10.1109/TNSM.2023.3318508`.

[34] Johannes Zerwas, Patrick Kalmbach, Stefan Schmid, and Andreas Blenk. "Ismael: Using Machine Learning To Predict Acceptance of Virtual Clusters in Data Centers". In: *IEEE TNSM* 16 (2019), pp. 950–964. DOI: `10.1109/TNSM.2019.2927291`.

[50] Anna Prado, Franziska Stöckeler, Fidan Mehmeti, Krämer Patrick, and Wolfgang Kellerer. "Enabling Proportionally-Fair Mobility Management with Reinforcement Learning in 5G Networks". In: *IEEE JSAC* 3GPP Technologies: 5G-Advanced and Beyond (2023).

Conference articles

[35] Johannes Zerwas et al. "NetBOA: Self-Driving Network Benchmarking". In: *NetAI '19*. Beijing, China: ACM, 2019. DOI: `10.1145/3341216.3342207`.

[36] Patrick Kalmbach, Andreas Blenk, Wolfgang Kellerer, Rastin Pries, Michael Jarschel, and Marco Hoffmann. "GPU Accelerated Planning and Placement of Edge Clouds". In: *NetSys 2019*. Munich, Germany: IEEE, 2019. DOI: `10.1109/NetSys.2019.8854495`.

[37] Patrick Kalmbach, Fabian Lipp, David Hock, Wolfgang Kellerer, and Andreas Blenk. "NOracle: Who is communicating with whom in my network?" In: *SIGCOMM '19 Posters and Demos*. Ed. by ACM. Beijing, China: ACM, Aug. 2019. DOI: `10.1145/3342280.3342303`.

[38] Andreas Blenk, Patrick Kalmbach, Johannes Zerwas, Michael Jarschel, Stefan Schmid, and Wolfgang Kellerer. “NeuroViNE: A Neural Preprocessor for Your Virtual Network Embedding Algorithm”. In: *INFOCOM 2018*. Honolulu, HI, USA: IEEE, Apr. 2018. DOI: `10.1109/INFOCOM.2018.8486263`.

[39] Patrick Kalmbach, Johannes Zerwas, Péter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. “Empowering Self-Driving Networks”. In: *SelfDN 2018*. Ed. by ACM. Budapest, Hungary: ACM, 2018. ISBN: 978-1-4503-5914-6. DOI: `10.1145/3229584.3229587`.

[40] Johannes Zerwas et al. “AHAB: Data-Driven Virtual Cluster Hunting”. In: *IFIP Networking 2018*. Ed. by IFIP. Zurich, Switzerland: IEEE, May 2018. ISBN: 978-3-903176-08-9. DOI: `10.23919/IFIPNetworking.2018.8696399`.

[41] Patrick Kalmbach, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. “Themis: Data Driven Approach to Botnet Detection”. In: *INFOCOM 2018 Posters and Demos*. Honolulu, Hawaii: IEEE, 2018.

[42] Andreas Blenk, Patrick Kalmbach, Johannes Zerwas, Wolfgang Kellerer, and Stefan Schmid. “Network Algorithms Ex Machina: Towards Self-Driving Networks”. In: *Cloud Control Workshop*. Skåvsjöholm, Schweden, June 2018.

[43] Andreas Blenk, Patrick Kalmbach, Stefan Schmid, and Wolfgang Kellerer. “o'zapft is: Tap Your Network Algorithm's Big Data!” In: *Big-DAMA '17*. Los Angeles, CA, USA, Aug. 2017.

[44] Mu He, Patrick Kalmbach, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. “Algorithm-Data Driven Optimization of Adaptive Communication Networks”. In: *ICNP 2017*. Toronto, ON, Canada: IEEE, Oct. 2017. DOI: `10.1109/ICNP.2017.8117592`.

[45] Patrick Kalmbach, Andreas Blenk, Markus Klügel, and Wolfgang Kellerer. “Generating Synthetic Internet- and IP-Topologies using the Stochastic-Block-Model”. In: *IM 2017*. Lisbon, Portugal: IEEE, May 2017. DOI: `10.23919/INM.2017.7987411`.

[46] Andreas Blenk, Patrick Kalmbach, Patrick van der Smagt, and Wolfgang Kellerer. “Boost Online Virtual Network Embedding: Using Neural Networks for Admission Control”. In: *CNSM*. IEEE, Oct. 2016. DOI: `10.1109/CNSM.2016.7818395`.

[47] Arsany Basta, Andreas Blenk, Laurenz Henkel, Patrick Kalmbach, Hassib Belhaj Hassine, and Wolfgang Kellerer. “HyperFlex: Towards Reliable and Dynamic SDN Virtualization for Next Generation Mobile Networks”. In: *48th Meeting of the VDE/ITG Section 5.2.4 "IP and Mobility" - 5G System*

Architecture. 48th Meeting of the VDE/ITG Section 5.2.4 "IP and Mobility" - 5G System Architecture, 2015.

[48] Patrick Krämer and Andreas Blenk. “Navigating Communication Networks with Deep Reinforcement Learning”. In: *NetSys 2021*. Vol. 80. Lübeck, Germany: ECEASST, Sept. 2021, p. 16.

[49] Alexander Griessel, Maximilian Stephan, Martin Mieth, Wolfgang Kellerer, and Patrick Krämer. “RLBrowse: Generating Realistic Packet Traces with Reinforcement Learning”. In: *NOMS 2022*. Budapest, Hungary: IEEE, 2022, pp. 1–6. DOI: 10.1109/NOMS54207.2022.9789851.

[51] Patrick Krämer, Philip Diederich, Corinna Krämer, Rastin Pries, Wolfgank Kellerer, and Andreas Blenk. “sfc2cpu: Operating a Service Function Chain Platform with Neural Combinatorial Optimization”. In: *IM 2021*. Bordeaux, France: IEEE, 2021, pp. 196–205.

Book chapters

[14] Andreas Blenk, Patrick Krämer, Johannes Zerwas, and Stefan Schmid. “Designing Algorithms for Data-Driven Network Management and Control: State-of-the-Art and Challenges”. In: *Communication Networks and Service Management in the Era of Artificial Intelligence and Machine Learning*. Section: 8. John Wiley & Sons, Ltd, 2021. ISBN: 978-1-119-67550-1. DOI: 10.1002/9781119675525.ch8.

Technical reports

[52] Patrick Krämer, Johannes Zerwas, and Andreas Blenk. *Mistill: Distilling Distributed Network Protocols from Examples*. 2022. URL: https://openreview.net/forum?id=gijKplIZ2Y-.

[119] Patrick Krämer et al. *A comprehensive dataset of website traffic*. PCAP. 2023. DOI: 10.14459/2023mp1700647.

[267] Patrick Krämer, Philip Diederich, Corinna Krämer, Rastin Pries, Wolfgang Kellerer, and Andreas Blenk. *Testbed measurements of co-located cloud network functions*. Unstructured. 2023. DOI: 10.14459/2023mp1707094.

General Publications

[1] Navid Nikaein, Eryk Schiller, Romain Favraud, Raymond Knopp, Islam Alyafawi, and Torsten Braun. "Towards a Cloud-Native Radio Access Network". In: *Advances in Mobile Cloud Computing and Big Data in the 5G Era*. Vol. 22. SBD. Cham, Swiss: Springer International Publishing AG, 2017, pp. 171–202. ISBN: 978-3-319-45145-9. DOI: 10.1007/978-3-319-45145-9_8.

[2] Nokia Corporation. *Innovate and execute with a simplified 5G core*. 2020. URL: https://onestore.nokia.com/asset/200888.

[3] *Cloud-native automation: The transformation of CSP networks*. 2023. URL: https://cloud.google.com/resources/cloud-native-automation-transformation-of-communications-service-provider-networks-research (visited on 03/18/2023).

[4] Pierre Imai, Paul Harvey, and Tareq Amin. *Towards a Truly Autonomus Network*. Tech. rep. Innovation Studio, Apr. 2020.

[5] ericsson. *Mobile data traffic outlook*. Jan. 2023. URL: https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/mobile-traffic-forecast (visited on 03/19/2023).

[6] Gerhard Fettweis et al. *The Tactile Internet*. Tech. rep. ITU&T Technology Watch, 2014. URL: https://www.itu.int/dms_pub/itu-t/oth/23/01/T23010000230001PDFE.pdf.

[7] ETSI. *Multi-access Edge Computing (MEC)*. 2021. URL: https://www.etsi.org/technologies/multi-access-edge-computing (visited on 11/28/2022).

[8] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. *Mobile Edge Computing A key technology towards 5G*. White Paper 11. ETSI, 2015. URL: https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf.

[9] Ajeet Das and Gary Chen. *Leveraging Cloud-Native Architecture for Agile Communications SP Networks*. Place: Needham, MA, USA. 2022.

[10] Jiao Zhang, F. Richard Yu, Shuo Wang, Tao Huang, Zengyi Liu, and Yunjie Liu. "Load Balancing in Data Center Networks: A Survey". In: *IEEE Commun. Surv. Tutorials* 20.3 (2018), pp. 2324–2352. DOI: 10.1109/COMST.2018.2816042.

[11] Deval Bhamare, Raj Jain, Mohammed Samaka, and Aiman Erbad. "A survey on service function chaining". In: *J. Netw. Comput. Appl.* 75 (2016), pp. 138–155. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.09.001.

[12] Enrique Dávalos and Benjamín Barán. "A Survey on Algorithmic Aspects of Virtual Optical Network Embedding for Cloud Networks". In: *IEEE Access* 6 (2018), pp. 20893–20906. DOI: 10.1109/ACCESS.2018.2821179.

[13] Nick Feamster and Jennifer Rexford. "Why (and How) Networks Should Run Themselves". In: *ANRW 2018*. Montreal, QC, Canada: ACM, 2018, p. 20. DOI: 10.1145/3232755.3234555.

[15] Natalia Vesselinova, Rebecca Steinert, Daniel F. Perez-Ramirez, and Magnus Boman. "Learning Combinatorial Optimization on Graphs: A Survey With Applications to Networking". In: *IEEE Access* 8 (2020), pp. 120388–120416. DOI: 10.1109/ACCESS.2020.3004964.

[16] Susanne Albers. "Online Scheduling". In: *Introduction to Scheduling*. Chapman and Hall/CRC Press, 2009, pp. 57–84.

[17] Maria-Florina Balcan. "Data-Driven Algorithm Design". In: *Beyond the Worst-Case Analysis of Algorithms*. Ed. by Tim Roughgarden. Cambridge University Press, 2020, pp. 626–645. DOI: 10.1017/9781108637435.036.

[18] Z. Zheng et al. "Octans: Optimal Placement of Service Function Chains in Many-Core Systems". In: *IEEE INFOCOM 2019*. Paris, France: IEEE, 2019, pp. 307–315. ISBN: 978-1-72810-515-4. DOI: 10.1109/INFOCOM.2019.8737544.

[19] R. Durner, C. Sieber, and W. Kellerer. "Towards Reducing Last-Level-Cache Interference of Co-Located Virtual Network Functions". In: *ICCCN*. Valencia, Spain: IEEE, 2019, pp. 1–9. DOI: 10.1109/ICCCN.2019.8846943.

[20] Christian Sieber, Raphael Durner, Maximilian Ehm, Wolfgang Kellerer, and Puneet Sharma. "Towards Optimal Adaptation of NFV Packet Processing to Modern CPU Memory Architectures". In: *CAN 17*. Incheon, Republic of Korea: ACM, 2017, pp. 7–12. ISBN: 978-1-4503-5423-3. DOI: 10.1145/3155921.3158429.

[21] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. "AuTO: Scaling Deep Reinforcement Learning for Datacenter-scale Automatic Traffic Optimization". In: *SIGCOMM '18*. Budapest, Hungary: ACM, 2018, pp. 191–205. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230551.

[22] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. "BLINC: multilevel traffic classification in the dark". In: *ACM SIGCOMM COMP COM*. Vol. 35. ACM, 2005, pp. 229–240. DOI: 10.1145/1090191.1080119. (Visited on 09/05/2016).

[23] Massimo Gallo, Alessandro Finamore, Gwendal Simon, and Dario Rossi. "FENXI: Fast In-Network Analytics". In: *SEC' 21*. San Jose, California, US: ACM, 2021.

[24] Petr Velan, Milan Čermák, Pavel Čeleda, and Martin Drašar. "A Survey of Methods for Encrypted Traffic Classification and Analysis". In: *Netw.* 25.5 (Sept. 2015), pp. 355–374. ISSN: 0028-3045.

[25] *5G-ready mobile network with virtualized network traffic analytics*. Tech. rep. 1. ipoque GmbH A Rohde & Schwarz Company, Apr. 2019. (Visited on 03/27/2020).

[26] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsanu Basta, Martin Reisslein, and Schmid Schmid. "Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges". In: *Proc. IEEE* 107.4 (2019), pp. 711–731. DOI: `10.1109/JPROC.2019.2895553`.

[27] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012. ISBN: 978-0-262-01802-9.

[28] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. "Machine learning for combinatorial optimization: A methodological tour d'horizon". In: *Eur. J. Oper. Res.* 290.2 (2021), pp. 405–421. DOI: `10.1016/j.ejor.2020.07.063`.

[32] Sean Michael Kerner. *New Gartner survey: Only half of AI models make it into production*. 2022. URL: `https://venturebeat.com/ai/new-gartner-survey-only-half-of-ai-models-make-it-into-production/` (visited on 11/09/2022).

[33] Venturebeat. *Why do 87% of data science projects never make it into production?* 2019. URL: `https://venturebeat.com/ai/why-do-87-of-data-science-projects-never-make-it-into-production/` (visited on 11/09/2022).

[53] Katie Costello and Meghan Rimol. *Gartner Identifies the Top Strategic Technology Trends for 2021*. 2020. URL: `https://www.gartner.com/en/newsroom/press-releases/2020-10-19-gartner-identifies-the-top-strategic-technology-trends-for-2021` (visited on 11/09/2022).

[54] Nadia Nahar, Shurui Zhou, Grace A. Lewis, and Christian Kästner. "Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process". In: *ICSE 2022*. Pittsburgh, PA, USA: ACM, 2022, pp. 413–425. DOI: `10.1145/3510003.3510209`.

[55] Lukas Fischer et al. "AI System Engineering - Key Challenges and Lessons Learned". In: *Mach. Learn. Knowl. Extr.* 3.1 (2021), pp. 56–83. DOI: `10.3390/make3010004`.

[56] Zhaoqi Xiong and Noa Zilberman. "Do Switches Dream of Machine Learning? Toward In-Network Classification". In: *HotNets '19*. Princeton, NJ, USA: ACM, 2019, pp. 25–33. ISBN: 978-1-4503-7020-2. DOI: `10.1145/3365609.3365864`.

[57] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. “Can the Network Be the AI Accelerator?” In: *NetCompute ’18*. NetCompute ’18. Budapest, Hungary: ACM, 2018, pp. 20–25. ISBN: 978-1-4503-5908-5. DOI: `10.1145/3229591.3229594`.

[58] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. “Taurus: An Intelligent Data Plane”. In: *CoRR* abs/2002.08987 (2020). arXiv: 2002.08987. URL: `https://arxiv.org/abs/2002.08987`.

[59] Giuseppe Siracusano and Roberto Bifulco. “In-network Neural Networks”. In: *CoRR* abs/1801.05731 (2018). URL: `http://arxiv.org/abs/1801.05731`.

[60] Nick Hotz. *What is CRISP DM?* 2022. URL: `https://www.datascience-pm.com/crisp-dm-2/` (visited on 10/28/2022).

[61] Marlon Dumas. *Fundamentals of business process management*. 2nd ed. Berlin: Springer, 2018. ISBN: 978-3-662-56509-4.

[62] R. Wirth and Jochen Hipp. “CRISP-DM: Towards a standard process model for data mining”. In: *Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining*. Manchester, UK, 2000, pp. 29–39.

[63] Colin Shearer. “The CRISP-DM Model: The New Blueprint for Data Mining”. In: *Journal of Data Warehousing* 5.4 (2000), pp. 13–22.

[64] Hajo Wiemer, Lucas Drowatzky, and Steffen Ihlenfeldt. “Data Mining Methodology for Engineering Applications (DMME)—A Holistic Extension to the CRISP-DM Model”. In: *Appl. Sci.* 9.12 (2019). ISSN: 2076-3417. DOI: `10.3390/app9122407`.

[65] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. “From Data Mining to Knowledge Discovery in Databases”. In: *AI Magazine* 17.3 (Mar. 1996), p. 37. DOI: `10.1609/aimag.v17i3.1230`.

[66] *Data Mining and the Case for Sampling*. Tech. rep. Cary, NC, USA: SAS Institute Inc., 1998.

[67] Veronika Plotnikova, Marlon Dumas, and Fredrik Milani. “Adaptations of data mining methodologies: a systematic literature review”. In: *PeerJ Comput. Sci.* 6 (2020), pp. 1–43. DOI: `10.7717/peerj-cs.267`.

[68] Fernando Martínez-Plumed et al. “CRISP-DM Twenty Years Later: From Data Mining Processes to Data Science Trajectories”. In: *IEEE Trans. Knowl. Data Eng.* 33.8 (2021), pp. 3048–3061. DOI: `10.1109/TKDE.2019.2962680`.

[69] Gregory Piatetsky, Ralph Winters, James Taylor, Martin Jetton, and Breno C. Costa. *CRISP-DM, still the top methodology for analytics, data mining, or data science projects*. Oct. 2014. URL: `https://www.kdnuggets.com/2014/10/crisp-dm-top-methodology-analytics-data-mining-data-science-projects.html` (visited on 11/21/2022).

[70] Stefan Studer et al. “Towards CRISP-ML(Q): A Machine Learning Process Model with Quality Assurance Methodology”. In: *Mach. Learn. Knowl. Extr.* 3.2 (2021), pp. 392–413. DOI: 10.3390/make3020020.

[71] Helge Toutenburg and Helge Knöfel. *Six Sigma - Methoden und Statistik für die Praxis*. 2nd ed. doi: 10.1007/978-3-540-85138-7. Heidelberg, Germany: Springer Berlin, 2009. ISBN: 978-3-540-85138-7.

[72] Zhongyi Pei, Lin Liu, Chen Wang, and Jianmin Wang. “Requirements Engineering for Machine Learning: A Review and Reflection”. In: Melbourne, Australia, 2022, pp. 166–175. DOI: 10.1109/REW56159.2022.00039.

[73] “IEEE Standard for Developing Software Life Cycle Processes”. In: *IEEE Std 1074-1995* (1996), pp. 1–106. DOI: 10.1109/IEEESTD.1996.79663.

[74] Gion Reto Cantieni, Gianluca Iannaccone, Chadi Barakat, Christophe Diot, and Patrick Thiran. “Reformulating the monitor placement problem: optimal network-wide sampling”. In: *CoNEXT '06*. Lisboa, Portugal: ACM, 2006, p. 5. DOI: 10.1145/1368436.1368444.

[75] Amaury Van Bemten, Nemanja Deric, Amir Varasteh, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. “Empirical Predictability Study of SDN Switches”. In: *ANCS 2019*. Cambridge, UK: IEEE, 2019, pp. 1–13. DOI: 10.1109/ANCS.2019.8901878.

[76] Johannes Zerwas et al. “NetBOA: Self-Driving Network Benchmarking”. In: *NetAI '19*. Beijing, China: ACM, 2019, pp. 8–14. DOI: 10.1145/3341216.3342207.

[77] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. “High-resolution Measurement of Data Center Microbursts”. In: *IMC '17*. London, United Kingdom: ACM, 2017, pp. 78–85. ISBN: 978-1-4503-5118-8. DOI: 10.1145/3131365.3131375.

[78] Silvery Fu, Saurabh Gupta, Radhika Mittal, and Sylvia Ratnasamy. “On the Use of ML for Blackbox System Performance Prediction”. In: *NSDI 21*. USENIX Association, Apr. 2021, pp. 763–784. ISBN: 978-1-939133-21-2.

[79] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data mining: practical machine learning tools and techniques, 3rd Edition*. Morgan Kaufmann, Elsevier, 2011. ISBN: 9780123748560.

[80] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009. ISBN: 978-0262013192.

[81] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, MA: MIT Press, 2018. ISBN: 978-0-262-03924-6.

[82] S Brandner and J MacQuaid. *Benchmarking Methodology for Network Interconnect Devices*. RFC 2544. RFC Editor, Mar. 1999, p. 31. DOI: 10.17487/RFC2544. URL: https://www.rfc-editor.org/info/rfc2544.

[83] Nipuni Hewage and Dulani Meedeniya. "Machine Learning Operations: A Survey on MLOps Tool Support". In: *CoRR* abs/2202.10169 (2022). URL: `https://arxiv.org/abs/2202.10169`.

[84] Richard Barnes et al. *Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement*. RFC 7624. Aug. 2015. DOI: `10.17487/RFC7624`. URL: `https://www.rfc-editor.org/info/rfc7624`.

[85] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. *Specification for DNS over Transport Layer Security (TLS)*. RFC 7858. May 2016. DOI: `10.17487/RFC7858`. URL: `https://www.rfc-editor.org/info/rfc7858`.

[86] Paul E. Hoffman and Patrick McManus. *DNS Queries over HTTPS (DoH)*. RFC 8484. Oct. 2018. DOI: `10.17487/RFC8484`. URL: `https://www.rfc-editor.org/info/rfc8484`.

[87] Christian Huitema and Eric Rescorla. *Issues and Requirements for Server Name Identification (SNI) Encryption in TLS*. RFC 8744. July 2020. DOI: `10.17487/RFC8744`. URL: `https://www.rfc-editor.org/info/rfc8744`.

[88] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-13. Work in Progress. Internet Engineering Task Force, Aug. 2021. 48 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-13`.

[89] Stephan Friedl, Andrei Popov, Adam Langley, and Stephan Emile. *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension*. RFC 7301. July 2014. DOI: `10.17487/RFC7301`. URL: `https://www.rfc-editor.org/info/rfc7301`.

[90] Zimo Chai, Amirhossein Ghafari, and Amir Houmansadr. "On the Importance of Encrypted-SNI (ESNI) to Censorship Circumvention". In: *FOCI 19*. Santa Clara, CA: USENIX Association, Aug. 2019.

[91] Marc Liberatore and Brian Neil Levine. "Inferring the Source of Encrypted HTTP Connections". In: *CCS '06*. Alexandria, Virginia, USA: ACM, 2006, pp. 255–263. ISBN: 1-59593-518-5. DOI: `10.1145/1180405.1180437`.

[92] S. E. Coull, M. P. Collins, C. V. Wright, F. Monrose, and M. K. Reiter. "On Web Browsing Privacy in Anonymized NetFlows". In: *SS'07*. Boston, MA, USA: USENIX Association, 2007, pp. 1–14.

[93] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. "Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier". In: *CCSW '09*. Chicago, Illinois, USA: ACM, 2009, pp. 31–42. ISBN: 978-1-60558-784-4. DOI: `10.1145/1655008.1655013`.

[94] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. "Website Fingerprinting in Onion Routing Based Anonymization Networks". In: *WPES '11*. Chicago, Illinois, USA: ACM, 2011, pp. 103–114. ISBN: 978-1-4503-1002-4. DOI: 10.1145/2046556.2046570.

[95] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. "Touching from a Distance: Website Fingerprinting Attacks and Defenses". In: *CCS '12*. Raleigh, North Carolina, USA: ACM, 2012, pp. 605–616. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382260.

[96] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail". In: *SSP*. San Francisco, CA, USA: IEEE, 2012, pp. 332–346. DOI: 10.1109/SP.2012.28.

[97] Tao Wang and Ian Goldberg. "Improved Website Fingerprinting on Tor". In: *WPES '13*. Berlin, Germany: ACM, 2013, pp. 201–212. ISBN: 978-1-4503-2485-4. DOI: 10.1145/2517840.2517851.

[98] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. "Effective Attacks and Provable Defenses for Website Fingerprinting". In: *SEC'14*. San Diego, CA: USENIX Association, 2014, pp. 143–157. ISBN: 978-1-931971-15-7.

[99] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. "A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses". In: *CCS '14*. Scottsdale, AZ, USA: ACM, 2014, pp. 227–238. DOI: 10.1145/2660267.2660362.

[100] Xiaodan Gu, Ming Yang, and Junzhou Luo. "A novel Website Fingerprinting attack against multi-tab browsing behavior". In: *CSCWD*. Calabria, Italy: IEEE, 2015, pp. 234–239. DOI: 10.1109/CSCWD.2015.7230964.

[101] Tao Wang and Ian Goldberg. "On Realistically Attacking Tor with Website Fingerprinting". In: *Proc. Priv. Enh.* 2016.4 (2016), pp. 21–36. DOI: doi : 10.1515/popets-2016-0027.

[102] Andriy Panchenko et al. "Website Fingerprinting at Internet Scale". In: *NDSS*. San Diego, California, USA: The Internet Society, 2016.

[103] Jamie Hayes and George Danezis. "K-Fingerprinting: A Robust Scalable Website Fingerprinting Technique". In: *SEC'16*. Austin, TX, USA: USENIX Association, 2016, pp. 1187–1203. ISBN: 978-1-931971-32-4.

[104] Yixiao Xu, Tao Wang, Qi Li, Qingyuan Gong, Yang Chen, and Yong Jiang. "A Multi-Tab Website Fingerprinting Attack". In: *ACSAC '18*. San Juan, PR, USA: ACM, 2018, pp. 327–341. ISBN: 978-1-4503-6569-7. DOI: 10.1145/3274694.3274697.

[105] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. "Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning". In: *CCS '18*. Toronto, Canada: ACM, 2018, pp. 1928–1943. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243768. URL: https://doi.org/10.1145/3243734.3243768.

[106] Payap Sirinam, Nate Mathews, Mohammad Saidur Rahman, and Matthew Wright. "Triplet Fingerprinting: More Practical and Portable Website Fingerprinting with N-Shot Learning". In: *CCS '19*. London, United Kingdom: ACM, 2019, pp. 1131–1148. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354217.

[107] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. "Automated Website Fingerprinting through Deep Learning". In: *NDSS 2018*. San Diego, CA, USA: The Internet Society, 2018.

[108] Zhongliu Zhuo, Yang Zhang, Zhi-Li Zhang, Xiaosong Zhang, and Jingzhong Zhang. "Website Fingerprinting Attack on Anonymity Networks Based on Profile Hidden Markov Model". In: *IEEE Trans. Inf. Forensics Secur.* 13.5 (2018), pp. 1081–1095. DOI: 10.1109/TIFS.2017.2762825.

[109] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. "Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning". In: *Proc. Priv. Enhancing Technol.* 2019.4 (2019), pp. 292–310. DOI: 10.2478/popets-2019-0070.

[110] Meng Shen, Yiting Liu, Siqi Chen, Liehuang Zhu, and Yuchao Zhang. "Webpage Fingerprinting using Only Packet Length Information". In: *ICC 2019*. Shanghai, China: IEEE, 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761167.

[111] Weiqi Cui, Tao Chen, and Eric Chan-Tin. "More Realistic Website Fingerprinting Using Deep Learning". In: *ICDCS 2020*. Singapore, Singapore: IEEE, 2020, pp. 333–343. DOI: 10.1109/ICDCS47774.2020.00058.

[112] Nguyen Phong Hoang, Arian Akhavan Niaki, Phillipa Gill, and Michalis Polychronakis. "Domain name encryption is not enough: privacy leakage via IP-based website fingerprinting". In: *Proc. Priv. Enhancing Technol.* 2021.4 (2021), pp. 420–440. DOI: 10.2478/popets-2021-0078.

[113] Tobias Pulls and Rasmus Dahlberg. "Website Fingerprinting with Website Oracles". In: *Proc. Priv. Enhancing Technol.* 2020.1 (2020), pp. 235–255. DOI: 10.2478/popets-2020-0013.

[114] Tao Wang. "High Precision Open-World Website Fingerprinting". In: *SP 2020*. San Francisco, CA, USA: IEEE, 2020, pp. 152–167. DOI: 10.1109/SP40000.2020.00015.

[115] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. “Statistical Identification of Encrypted Web Browsing Traffic”. In: *S&P’02*. Berkeley, CA, USA: IEEE Computer Society, 2002, pp. 19–30. DOI: 10.1109/SECPRI.2002.1004359.

[116] Kota Abe and Shigeki Goto. “Fingerprinting Attack on Tor Anonymity using Deep Learning”. In: *APAN 2016*. Hong Kong, China: Asia-Pacific Advanced Network Ltd., 2016. ISBN: 978-4-9905448-6-7.

[117] Martino Trevisan, Francesca Soro, Marco Mellia, Idilio Drago, and Ricardo Morla. “Does Domain Name Encryption Increase Users’ Privacy?” In: *SIGCOMM Comput. Commun. Rev.* 50.3 (July 2020). Place: New York, NY, USA Publisher: ACM. ISSN: 0146-4833. DOI: 10.1145/3411740.3411743.

[118] Weiqi Cui, Tao Chen, Christian Fields, Julianna Chen, Anthony Sierra, and Eric Chan-Tin. “Revisiting Assumptions for Website Fingerprinting Attacks”. In: *AsiaCCS’ 2019*. Auckland, New Zealand: ACM, 2019, pp. 328–339. DOI: 10.1145/3321705.3329802.

[120] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. “Internet Inter-Domain Traffic”. In: *SIGCOMM’ 10*. New Delhi, India: ACM, 2010, pp. 75–86. DOI: 10.1145/1851182.1851194.

[121] Bernhard Ager, Wolfgang Mühlbauer, Georgios Smaragdakis, and Steve Uhlig. “Web Content Cartography”. In: *IMC ’11*. Berlin, Germany, 2011, pp. 585–600. DOI: 10.1145/2068816.2068870. URL: http://doi.acm.org/10.1145/2068816.2068870.

[122] Timm Böttger, Félix Cuadrado, and Steve Uhlig. “Looking for Hypergiants in PeeringDB”. In: *Computer Communication Review* 48.3 (2018), pp. 13–19. DOI: 10.1145/3276799.3276801.

[123] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. “Blocking-resistant communication through domain fronting”. In: *Proc. Priv. Enhancing Technol.* 2015.2 (June 2015), pp. 46–64. DOI: 10.1515/popets-2015-0009.

[124] Pijus Jauniškis. *VPN statistics: Users, markets, & legality*. Mar. 2022. URL: https://surfshark.com/blog/vpn-users (visited on 04/19/2022).

[125] Kevin Bock, Louis-Henri Merino, David Fifield, Amir Houmansadr, and Dave Levin. *Exposing and Circumventing China’s Censorship of ESNI*. July 2020. URL: https://gfw.report/blog/gfw%5C_esni%5C_blocking/en/ (visited on 04/15/2022).

[126] Kleidi Ismailaj, Miguel Camelo, and Steven Latré. “When Deep Learning May Not Be The Right Tool For Traffic Classification”. In: *IM 2021*. Bordeaux, France: IEEE, May 2021, pp. 884–889.

[127] Robert Geirhos et al. “Shortcut learning in deep neural networks”. In: *Nature Machine Intelligence* 2.11 (Nov. 2020), pp. 665–673. ISSN: 2522-5839. DOI: 10.1038/s42256-020-00257-z.

[128] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018, pp. 1–159. URL: https://datatracker.ietf.org/doc/html/rfc8446.

[129] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M. Maggs. “On Landing and Internal Web Pages: The Strange Case of Jekyll and Hyde in Web Performance Measurement”. In: *IMC '20*. Virtual Event, USA: ACM, 2020, pp. 680–695. ISBN: 978-1-4503-8138-3. DOI: 10.1145/3419394.3423626.

[130] Lawrence R. Rabiner. “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proc. IEEE* 77.2 (1989), pp. 257–286. DOI: 10.1109/5.18626.

[131] Andrew Hintz. “Fingerprinting Websites Using Traffic Analysis”. In: *PET 2002*. Vol. 2482. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, 2002, pp. 171–178. DOI: 10.1007/3-540-36467-6_13.

[132] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. “A Critical Evaluation of Website Fingerprinting Attacks”. In: *CCS '14*. Scottsdale, Arizona, USA: ACM, 2014, pp. 263–274. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660368.

[133] Xun Gong, Negar Kiyavash, and Nikita Borisov. “Fingerprinting websites using remote traffic analysis”. In: *CCS '10*. Chicago, Illinois, USA: ACM, 2010, pp. 684–686. DOI: 10.1145/1866307.1866397.

[134] Marc Juárez, Mohsen Imani, Mike Perry, Claudia Díaz, and Matthew Wright. “Toward an Efficient Website Fingerprinting Defense”. In: *ESORICS 2016*. Vol. 9878. Heraklion, Greece: Springer, 2016, pp. 27–46. DOI: 10.1007/978-3-319-45744-4_2.

[135] Tao Wang and Ian Goldberg. “Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks”. In: *USENIX Security 17*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1375–1390. ISBN: 978-1-931971-40-9.

[136] David Lu, Sanjit Bhat, Albert Kwon, and Srinivas Devadas. “DynaFlow: An Efficient Website Fingerprinting Defense Based on Dynamically-Adjusting Flows”. In: *WPES'18*. WPES'18. Toronto, Canada: ACM, 2018, pp. 109–113. ISBN: 978-1-4503-5989-4. DOI: 10.1145/3267323.3268960.

[137] Wladimir De la Cadena et al. “TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting”. In: CCS '20. Virtual Event, USA: ACM, 2020, pp. 1971–1985. ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3423351.

[138] Meier, Roland, Lenders, Vinvent, and Vanbever, Laurent. “ditto: WAN Traffic Obfuscation at Line Rate”. In: *NDSS 2022*. San Diego, CA, USA: Internet Society, Apr. 2022.

[139] Charles V. Wright, Scott E. Coull, and Fabian Monrose. “Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis”. In: *NDSS 2009*. San Diego, CA, USA: The Internet Society, Feb. 2009.

[140] Giovanni Cherubin, Jamie Hayes, and Marc Juárez. “Website Fingerprinting Defenses at the Application Layer”. In: *Proc. Priv. Enhancing Technol.* 2017.2 (2017), pp. 186–203. DOI: `10.1515/popets-2017-0023`.

[141] Eric Chan-Tin, Taejoon Kim, and Jinoh Kim. “Website Fingerprinting Attack Mitigation Using Traffic Morphing”. In: *ICDCS 2018*. IEEE Computer Society, 2018, pp. 1575–1578. DOI: `10.1109/ICDCS.2018.00174`.

[142] Shui Yu, Guofeng Zhao, Wanchun Dou, and Simon James. “Predicted Packet Padding for Anonymous Web Browsing Against Traffic Analysis Attacks”. In: *IEEE Trans. Inf. Forensics Secur.* 7.4 (2012), pp. 1381–1393. DOI: `10.1109/TIFS.2012.2197392`.

[143] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. “SkypeMorph: protocol obfuscation for Tor bridges”. In: *CCS'12*. li2012. Raleigh, NC, USA: ACM, Oct. 2012, pp. 97–108. DOI: `10.1145/2382196.2382210`.

[144] Weiqi Cui, Jiangmin Yu, Yanmin Gong, and Eric Chan-Tin. “Realistic Cover Traffic to Mitigate Website Fingerprinting Attacks”. In: *ICDCS 2018*. IEEE Computer Society, 2018, pp. 1579–1584. DOI: `10.1109/ICDCS.2018.00175`.

[145] Xiapu Luo, Peng Zhou, Edmond W. W. Chan, Wenke Lee, Rocky K. C. Chang, and Roberto Perdisci. “HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows”. In: *NDSS 2011*. San Diego, CA, USA: The Internet Society, 2011.

[146] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y. Zhao. “A Real-time Defense against Website Fingerprinting Attacks”. In: *CoRR* abs/2102.04291 (2021). arXiv: 2102.04291. URL: `https://arxiv.org/abs/2102.04291`.

[147] Jean Paul Degabriele. “Hiding the Lengths of Encrypted Messages via Gaussian Padding”. In: *CCS'21*. Virtual Event, Republic of Korea: ACM, 2021, pp. 1549–1565. DOI: `10.1145/3460120.3484590`.

[148] Xiaolei Liu, Zhongliu Zhuo, Xiaojiang Du, Xiaosong Zhang, Qingxin Zhu, and Mohsen Guizani. “Adversarial attacks against profile HMM website fingerprinting detection model”. In: *Cognitive Systems Research* 54 (2019), pp. 83–89. ISSN: 1389-0417. DOI: `https://doi.org/10.1016/j.cogsys.2018.12.005`.

[149] Jiajun Gong and Tao Wang. "Zero-delay Lightweight Defenses against Website Fingerprinting". In: *Security '20*. Virtual Event: USENIX Association, 2020, pp. 717–734.

[150] Rishab Nithyanand, Xiang Cai, and Rob Johnson. "Glove: A Bespoke Website Fingerprinting Defense". In: *WPES '14*. Scottsdale, AZ, USA: ACM, 2014, pp. 131–134. DOI: 10.1145/2665943.2665950.

[151] Thuy T.T. Nguyen and Grenville Armitage. "A survey of techniques for internet traffic classification using machine learning". In: *EEE Commun. Surv. Tutor.* 10.4 (2008), pp. 56–76. ISSN: 1553-877X. DOI: 10.1109/SURV.2008.080406.

[152] Hyunchul Kim, KC Claffy, Marina Fomenkov, Dhiman Barman, Michalis Faloutsos, and KiYoung Lee. "Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices". In: *CoNEXT '08*. Madrid, Spain: ACM, 2008, 11:1–11:12. ISBN: 978-1-60558-210-8. DOI: 10.1145/1544012.1544023.

[153] Riyad Alshammari and A. Nur Zincir-Heywood. "Can encrypted traffic be identified without port numbers, IP addresses and payload inspection?" In: *Comput. Netw* 55.6 (2011), pp. 1326–1350. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2010.12.002.

[154] Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. "AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic". In: *EuroS&P*. Mar. 2016, pp. 439–454. DOI: 10.1109/EuroSP.2016.40.

[155] Iman Akbari et al. "A Look Behind the Curtain: Traffic Classification in an Increasingly Encrypted Web". In: *Proc. ACM Meas. Anal. Comput. Syst.* 5.1 (Feb. 2021). DOI: 10.1145/3447382.

[156] M. Korczyński and A. Duda. "Markov chain fingerprinting to classify encrypted traffic". In: *INFOCOM 2014*. Toronto, ON, Canada: IEEEE, Apr. 2014, pp. 781–789. DOI: 10.1109/INFOCOM.2014.6848005.

[157] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. "Programmable Switches for in-Networking Classification". In: *IEEE INFOCOM 2021*. Vancouver, BC, Canada: IEEE, 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488840.

[158] Fannia Pacheco, Ernesto Exposito, Mathieu Gineste, Cedric Baudoin, and Jose Aguilar. "Towards the Deployment of Machine Learning Solutions in Network Traffic Classification: A Systematic Survey". In: *IEEE Communications Surveys Tutorials* 21.2 (2019), pp. 1988–2014. DOI: 10.1109/COMST.2018.2883147.

[159] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunström. "Measuring Latency Variation in the Internet". In: *CoNEXT '16*. Irvine, CA, USA: ACM, 2016, pp. 473–480. DOI: 10.1145/2999572.2999603.

[160] Anders Krogh, Michael Brown, I. Saira Mian, Kimmen Sjölander, and David Haussler. "Hidden Markov Models in Computational Biology: Applications to Protein Modeling". In: *J. Mol. Biol.* 235.5 (1994), pp. 1501–1531. ISSN: 0022-2836. DOI: https://doi.org/10.1006/jmbi.1994.1104.

[161] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. "On Inferring Application Protocol Behaviors in Encrypted Network Traffic". In: vol. 7. JMLR.org, Dec. 2006, pp. 2745–2769. DOI: 10.5555/1248547.1248647.

[162] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau-Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press, 2009. ISBN: 6130216904.

[163] *Alexa - The Web Information Company*. URL: https://www.alexa.com/ (visited on 05/14/2020).

[164] Hasan Faik Alan and Jasleen Kaur. "Client Diversity Factor in HTTPS Webpage Fingerprinting". In: *CODASPY '19*. Richardson, TX, USA: ACM, 2019, pp. 279–290. ISBN: 978-1-4503-6099-9. DOI: 10.1145/3292006.3300045.

[165] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (2014). ISSN: 1075-3583.

[166] Hanlin Chen, Hongmei He, and Andrew Starr. "An Overview of Web Robots Detection Techniques". In: *Cyber Security*. Dublin, Ireland: IEEE, 2020, pp. 1–6. DOI: 10.1109/CyberSecurity49315.2020.9138856.

[167] W John, M Dusi, and k claffy k. *Estimating Routing Symmetry on Single Links by Passive Flow Measurements*. Tech. rep. ACM TRAC, Mar. 2010.

[168] Roger Weber, Hans-Jörg Schek, and Stephen Blott. "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces". In: *VLDB'98*. New York, NY, USA: Morgan Kaufmann, 1998, pp. 194–205.

[169] Wei Zhang et al. "OpenNetVM: A Platform for High Performance Network Service Chains". In: *HotMIddlebox '16*. Florianopolis, Brazil: ACM, 2016, pp. 26–31. ISBN: 978-1-4503-4424-1. DOI: 10.1145/2940147.2940155.

[170] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. "MoonGen: A Scriptable High-Speed Packet Generator". In: *IMC '15*. Tokyo, Japan: Association for Computing Machinery, 2015, pp. 275–287. ISBN: 978-1-4503-3848-6. DOI: 10.1145/2815675.2815692.

[171] Theophilus Benson, Aditya Akella, and David A. Maltz. “Network Traffic Characteristics of Data Centers in the Wild”. In: *IMC ’10*. Melbourne, Australia: ACM, 2010, pp. 267–280. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879175.

[172] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. “Inside the Social Network’s (Datacenter) Network”. In: *SIGCOMM Comput. Commun. Rev.* 45.4 (Aug. 2015), pp. 123–137. ISSN: 0146-4833. DOI: 10.1145/2829988.2787472.

[173] Nandita Dukkipati and Nick McKeown. “Why Flow-completion Time is the Right Metric for Congestion Control”. In: *SIGCOMM Comput. Commun. Rev.* 36.1 (Jan. 2006), pp. 59–62. ISSN: 0146-4833. DOI: 10.1145/1111322.1111336.

[174] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. “Efficient Coflow Scheduling with Varys”. In: *SIGCOMM ’14*. Chicago, IL, USA: ACM, Aug. 2014, pp. 443–454. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626315.

[175] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. “HULA: Scalable Load Balancing Using Programmable Data Planes”. In: *SOSR ’16*. Santa Clara, CA, USA: ACM, 2016, 10:1–10:12. ISBN: 978-1-4503-4211-7. DOI: 10.1145/2890955.2890968.

[176] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. “MicroTE: Fine Grained Traffic Engineering for Data Centers”. In: *CoNEXT ’11*. Tokyo, Japan: ACM, 2011, 8:1–8:12. ISBN: 978-1-4503-1041-3. DOI: 10.1145/2079296.2079304.

[177] Albert Greenberg et al. “VL2: A Scalable and Flexible Data Center Network”. In: *SIGCOMM ’09*. Barcelona, Spain: ACM, 2009, pp. 51–62. ISBN: 978-1-60558-594-9. DOI: 10.1145/1592568.1592576.

[178] Srikanth Kandula, Sudipta Sengupta, Albert G. Greenberg, Parveen Patel, and Ronnie Chaiken. “The nature of data center traffic: measurements & analysis”. In: *IMC ’09*. Chicago, IL, USA: ACM, 2009, pp. 202–208. DOI: 10.1145/1644893.1644918.

[179] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. “Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching”. In: *NSDI 17*. Boston, MA, USA: USENIX Association, Mar. 2017, pp. 407–420. ISBN: 978-1-931971-37-9.

[180] Mohammad Alizadeh et al. “CONGA: Distributed Congestion-Aware Load Balancing for Datacenters”. In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 503–514. ISSN: 0146-4833. DOI: 10.1145/2740070.2626316.

[181] Naga Katta, Mukesh Hira, Aditi Ghag, Changhoon Kim, Isaac Keslassy, and Jennifer Rexford. "CLOVE: How I Learned to Stop Worrying About the Core and Love the Edge". In: *HotNets '16*. Atlanta, GA, USA: ACM, 2016, pp. 155–161. ISBN: 978-1-4503-4661-0. DOI: `10.1145/3005745.3005751`.

[182] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. "Contra: A Programmable System for Performance-aware Routing". In: *NSDI 20*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 701–721. ISBN: 978-1-939133-13-7.

[183] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *SIGCOMM Comput. Commun. Rev.* 38.4 (Aug. 2008), pp. 63–74. ISSN: 0146-4833. DOI: `10.1145/1402946.1402967`.

[184] Dinesh G. Dutt. *BGP in the Data Center*. O'Reilly Media, Inc., 2017. ISBN: 9781491983409.

[185] Ashish Vaswani et al. "Attention is All you Need". In: *NIPS 2017*. Long Beach, CA, USA: Curran Associates, Inc., 2017, pp. 6000–6010.

[186] Mohammad Noormohammadpour and Cauligi S Raghavendra. "Datacenter traffic control: Understanding techniques and tradeoffs". In: *IEEE Communications Surveys & Tutorials* 20.2 (2017), pp. 1492–1525.

[187] Marco Chiesa, Gábor Rétvári, and Michael Schapira. "Lying Your Way to Better Traffic Engineering". In: *CoNEXT '16*. Irvine, California, USA: ACM, 2016, pp. 391–398. DOI: `10.1145/2999572.2999585`.

[188] Nithin Michael and Ao Tang. "HALO: Hop-by-Hop Adaptive Link-State Optimal Routing". In: *IEEE/ACM Trans. Netw.* 23.6 (2015), pp. 1862–1875. DOI: `10.1109/TNET.2014.2349905`.

[189] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *NSDI 10*. San Jose, CA, USA: ACM, 2010, pp. 281–296.

[190] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. "CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark". In: *SIGCOMM '16*. Florianopolis, Brazil: ACM, 2016, pp. 160–173. ISBN: 978-1-4503-4193-6. DOI: `10.1145/2934872.2934880`.

[191] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. "Dynamic Load Balancing Without Packet Reordering". In: *SIGCOMM Comput. Commun. Rev.* 37.2 (Mar. 2007), pp. 51–62. ISSN: 0146-4833. DOI: `10.1145/1232919.1232925`.

[192] Giuseppe Siracusano and Roberto Bifulco. "In-network Neural Networks". In: *CoRR* abs/1801.05731 (2018). URL: `http://arxiv.org/abs/1801.05731`.

[193] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. “Challenging the Stateless Quo of Programmable Switches”. In: *HotNets ’20*. HotNets ’20. Virtual Event, USA: ACM, 2020, pp. 153–159. ISBN: 978-1-4503-8145-1. DOI: 10.1145/3422604.3425928.

[194] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. “Neural packet classification”. In: *SIGCOMM ’19*. Beijing, China: ACM, 2019, pp. 256–269. DOI: 10.1145/3341302.3342221.

[195] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. “A Computational Approach to Packet Classification”. In: *SIGCOMM ’20*. Virtual Event, USA: ACM, 2020, pp. 542–556. ISBN: 978-1-4503-7955-7. DOI: 10.1145/3387514.3405886.

[196] Shunsuke Higuchi, Yuki Koizumi, Junji Takemasa, Atsushi Tagami, and Toru Hasegawa. “Learned FIB: Fast IP Forwarding without Longest Prefix Matching”. In: *ICNP 2021*. Dallas, TX, USA: IEEE, 2021, pp. 1–11. DOI: 10.1109/ICNP52444.2021.9651956.

[197] Raouf Boutaba et al. “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities”. In: *J. Internet Serv. Appl.* 9.1 (June 2018), p. 16. ISSN: 1869-0238. DOI: 10.1186/s13174-018-0087-2.

[198] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. “A Machine Learning Approach to Routing”. In: *CoRR* abs/1708.03074 (2017). URL: http://arxiv.org/abs/1708.03074.

[199] B. Mao et al. “Routing or Computing? The Paradigm Shift Towards Intelligent Computer Network Packet Transmission Based on Deep Learning”. In: *IEEE TNSM* 66.11 (Nov. 2017), pp. 1946–1960. ISSN: 0018-9340. DOI: 10.1109/TC.2017.2709742.

[200] Ke Liang and Mitchel Myers. “Machine Learning Applications in the Routing in Computer Networks”. In: *CoRR* abs/2104.01946 (2021). arXiv: 2104.01946. URL: https://arxiv.org/abs/2104.01946.

[201] Abhiram Singh, Sidharth Sharma, and Ashwin Gumaste. “Using Deep Reinforcement Learning for Routing in IP Networks”. In: *ICCCN 2021*. Athens, Greece: IEEE, 2021, pp. 1–9. DOI: 10.1109/ICCCN52240.2021.9522197.

[202] Aleksandra Jereczek, Patrycja Kochmańska, and Maciej Paczkowski. “Machine learning in packet routing process using Quagga/Zebra routing SW suite”. In: *Netdev 0x14*. Virtual Event: The Netdev Society, July 2020. URL: https://legacy.netdevconf.info/0x14/pub/papers/49/0x14-paper49-talk-paper.pdf (visited on 09/02/2022).

[203] Bo Chen, Di Zhu, Yuwei Wang, and Peng Zhang. “An Approach to Combine the Power of Deep Reinforcement Learning with a Graph Neural Network for Routing Optimization”. In: *Electronics* 11.3 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11030368.

[204] Paul Almasan, José Suárez-Varela, Arnau Badia-Sampera, Krzysztof Rusek, Pere Barlet-Ros, and Albert Cabellos-Aparicio. “Deep Reinforcement Learning meets Graph Neural Networks: An optical network routing use case”. In: *CoRR* abs/1910.07421 (2019). URL: http://arxiv.org/abs/1910.07421.

[205] Zhiyuan Xu et al. “Experience-driven Networking: A Deep Reinforcement Learning based Approach”. In: *INFOCOM 2018*. Honolulu, HI, USA: IEEE, 2018, pp. 1871–1879. DOI: 10.1109/INFOCOM.2018.8485853.

[206] Penghao Sun, Yuxiang Hu, Julong Lan, Le Tian, and Min Chen. “TIDE: Time-Relevant Deep Reinforcement Learning for Routing Optimization”. In: *Future Gener. Comput. Syst.* 99.C (Oct. 2019). Place: NLD Publisher: Elsevier Science Publishers B. V., pp. 401–409. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.04.014.

[207] Tiansi Hu and Yunsi Fei. “QELAR: A Machine-Learning-Based Adaptive Routing Protocol for Energy-Efficient and Lifetime-Extended Underwater Sensor Networks”. In: *IEEE Trans. Mob. Comput.* 9.6 (June 2010), pp. 796–809. ISSN: 1536-1233. DOI: 10.1109/TMC.2010.28.

[208] Qian Xu, Yifan Zhang, Kui Wu, Jianping Wang, and Kejie Lu. “Evaluating and Boosting Reinforcement Learning for Intra-Domain Routing”. In: *MASS 2019*. Monterey, CA, USA: IEEE, 2019, pp. 265–273. DOI: 10.1109/MASS.2019.00039.

[209] Syed Qaisar Jalil, Mubashir Husain Rehmani, and Stephan Chalup. “DQR: Deep Q-Routing in Software Defined Networks”. In: *IJCNN*. Glasgow, UK: IEEE, 2020, pp. 1–8. DOI: 10.1109/IJCNN48605.2020.9206767.

[210] Yuanyuan Cao, Bin Dai, Yijun Mo, and Yang Xu. “IQoR: An Intelligent QoS-aware Routing Mechanism with Deep Reinforcement Learning”. In: *LCN*. Sydney, NSW, Australia: IEEE, 2020, pp. 329–332. DOI: 10.1109/LCN48667.2020.9314768.

[211] Xinyu You, Xuanjie Li, Yuedong Xu, Hui Feng, Jin Zhao, and Huaicheng Yan. “Toward Packet Routing With Fully Distributed Multiagent Deep Reinforcement Learning”. In: *{IEEE} Trans. Syst. Man Cybern. Syst.* 52.2 (2022), pp. 855–868. DOI: 10.1109/TSMC.2020.3012832.

[212] Fabien Geyer and Georg Carle. “Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning”. In: *Big-DAMA ’18*. Big-DAMA ’18. event-place: Budapest, Hungary: ACM, 2018, pp. 40–45. ISBN: 978-1-4503-5904-7. DOI: 10.1145/3229607.3229610.

[213] Shihan Xiao, Haiyan Mao, Bo Wu, Wenjie Liu, and Fenglin Li. "Neural Packet Routing". In: *NetAI'20*. Virtual Event, USA: ACM, 2020, pp. 28–34. ISBN: 978-1-4503-8043-0. DOI: 10.1145/3405671.3405813.

[214] Patrick Krämer and Andreas Blenk. "Navigating Communication Networks with Deep Reinforcement Learning". In: *NetSys 2021*. Vol. 80. Lübeck, Germany: ECEASST, Sept. 2021, p. 16.

[215] Yangzhe Kong, Dmitry Petrov, Vilho Räisänen, and Alexander Ilin. "Path-Link Graph Neural Network for IP Network Performance Prediction". In: *IM 2021*. Bordeaux, France: IEEE, 2021, pp. 170–177. ISBN: 978-3-903176-32-4.

[216] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. "Unveiling the Potential of Graph Neural Networks for Network Modeling and Optimization in SDN". In: *SOSR '19*. SOSR '19. San Jose, CA, USA: ACM, 2019, pp. 140–151. ISBN: 978-1-4503-6710-3. DOI: 10.1145/3314148.3314357.

[217] Fabien Geyer. "DeepComNet: Performance Evaluation of Network Topologies Using Graph-Based Deep Learning". In: *Perform. Eval.* 130.C (Apr. 2019), pp. 1–16. ISSN: 0166-5316. DOI: 10.1016/j.peva.2018.12.003.

[218] Mowei Wang, Linbo Hui, Yong Cui, Ru Liang, and Zhenhua Liu. "xNet: Improving Expressiveness and Granularity for Network Modeling with Graph Neural Networks". In: *INFOCOM 2022*. London, United Kingdom: IEEE, 2022, pp. 2028–2037. DOI: 10.1109/INFOCOM48880.2022.9796726.

[219] Shihan Xiao, Dongdong He, and Zhibo Gong. "Deep-Q: Traffic-Driven QoS Inference Using Deep Generative Network". In: *NetAI'18*. Budapest, Hungary: ACM, 2018, pp. 67–73. ISBN: 978-1-4503-5911-5. DOI: 10.1145/3229543.3229549.

[220] Junfei Li, Penghao Sun, and Yuxiang Hu. "Traffic modeling and optimization in datacenters with graph neural network". In: *Comput. Netw.* 181 (2020), p. 107528. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2020.107528.

[221] Miquel Ferriol-Galmés et al. "RouteNet-Erlang: A Graph Neural Network for Network Performance Evaluation". In: *INFOCOM 2022*. London, United Kingdom: IEEE, 2022, pp. 2018–2027. DOI: 10.1109/INFOCOM48880.2022.9796944.

[222] Miquel Ferriol-Galmés et al. "Building a Digital Twin for network optimization using Graph Neural Networks". In: *Comput. Netw.* (2022), p. 109329. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2022.109329.

[223] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. "Understanding Lifecycle Management Complexity of Datacenter Topologies". In: *NSDI 19*. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 235–254. ISBN: 978-1-931971-49-2.

[224] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *AISTATS 2011*. Vol. 15. Fort Lauderdale, FL, USA: JMLR, Apr. 2011, pp. 315–323.

[225] Thomas Luinaud, Thibaut Stimpfling, Jeferson Santiago da Silva, Yvon Savaria, and J. M. Pierre Langlois. "Bridging the Gap: FPGAs as Programmable Switches". In: *HPSR 2020*. Newark, NJ, USA: IEEE, May 2020, pp. 1–7. DOI: `10.1109/HPSR48589.2020.9098978`.

[226] Intel Corporation. *Intel® Xeon® Prozessor E3-1270*. URL: `https://ark.intel.com/content/www/de/de/ark/products/52276/intel-xeon-processor-e31270-8m-cache-3-40-ghz.html` (visited on 09/15/2022).

[227] Ravaindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 978-0136175490.

[228] M. Chiesa, G. Kindler, and M. Schapira. "Traffic engineering with Equal-Cost-Multipath: An algorithmic perspective". In: *INFOCOM 2014*. Toronto, Canada: IEEE, Apr. 2014, pp. 1590–1598. DOI: `10.1109/TNET.2016.2614247`.

[229] Junlan Zhou et al. "WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers". In: *EuroSys '14*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 5:1–5:14. ISBN: 978-1-4503-2704-6. DOI: `10.1145/2592798.2592803`.

[230] Kuo-Feng Hsu, Praveen Tammana, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. "Adaptive Weighted Traffic Splitting in Programmable Data Planes". In: *SOSR '20*. SOSR '20. San Jose, CA, USA: ACM, 2020, pp. 103–109. ISBN: 978-1-4503-7101-8. DOI: `10.1145/3373360.3380841`.

[231] Eric Jang, Shixiang Gu, and Ben Poole. "Categorical Reparameterization with Gumbel-Softmax". In: *ICLR 2017*. Toulon, France: OpenReview.net, 2017. URL: `https://openreview.net/forum?id=rkE3y85ee`.

[232] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. "The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables". In: *ICLR 2017*. Toulon: OpenReview.net, 2017. URL: `https://openreview.net/forum?id=S1jE5L5gl`.

[233] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. "Scalable High Speed IP Routing Lookups". In: *SIGCOMM '97*. SIGCOMM '97. Cannes, France: ACM, 1997, pp. 25–36. ISBN: 089791905X. DOI: `10.1145/263105.263136`.

[234] William L. Hamilton, Rex Ying, and Jure Leskovec. “Representation Learning on Graphs: Methods and Applications”. In: *CoRR* abs/1709.05584 (2017). arXiv: 1709.05584. URL: http://arxiv.org/abs/1709.05584.

[235] Vinith Misra and Sumit Bhatia. “Bernoulli Embeddings for Graphs”. In: *CoRR* abs/1803.09211 (2018).

[236] Aditya Grover and Jure Leskovec. “Node2vec: Scalable Feature Learning for Networks”. In: *KDD ’16*. San Francisco, CA, USA: ACM, 2016, pp. 855–864. ISBN: 9781450342322. DOI: 10.1145/2939672.2939754.

[237] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *ICLR 2015*. San Diego, CA, USA: OpenReview.net, 2015.

[238] Liam Li et al. “Massively Parallel Hyperparameter Tuning”. In: *CoRR* abs/1810.05934 (2018). arXiv: 1810.05934. URL: http://arxiv.org/abs/1810.05934.

[239] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. “Tune: A Research Platform for Distributed Model Selection and Training”. In: *CoRR* abs/1807.05118 (2018). arXiv: 1807.05118. URL: http://arxiv.org/abs/1807.05118.

[240] Christina Delimitrou, Sriram Sankar, Aman Kansal, and Christos Kozyrakis. “ECHO: Recreating network traffic maps for datacenters with tens of thousands of servers”. In: *IISWC*. La Jolla, CA, USA: IEEE, 2012, pp. 14–24. DOI: 10.1109/IISWC.2012.6402896.

[241] Andre Martins and Ramon Astudillo. “From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification”. In: *ICML 2016*. Vol. 48. New York, New York, USA: JMLR.org, June 2016, pp. 1614–1623.

[242] Clarence Filsfils, Pablo Camarillo, John Leddy, Daniel Voyer, Satoru Matsushima, and Zhenbin Li. *Segment Routing over IPv6 (SRv6) Network Programming*. RFC 8986. Feb. 2021. DOI: 10.17487/RFC8986. URL: https://www.rfc-editor.org/info/rfc8986.

[243] Kevin Deierling. *What Is a DPU?* May 2020. URL: https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu-data-processing-unit/.

[244] Jakub Kicinski and Nicolaas Viljoen. *eBPF Hardware Offload to SmartNICs: cls bpf and XDP*. Tech. rep. Netronome Systems, Sept. 2017. URL: https://www.netronome.com/media/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf.

[245] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. “Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications”. In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3371038.

[246] David Thaler and Poorna Gaddehosur. *Making eBPF work on Windows*. Oct. 2021. URL: https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/ (visited on 09/16/2022).

[247] PyTorch Foundation. *Loading a PyTorch Model in C++*. URL: https://pytorch.org/tutorials/advanced/cpp_export.html (visited on 10/10/2022).

[248] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. "Understanding Host Network Stack Overheads". In: *SIGCOMM '21*. SIGCOMM '21. Virtual Event, USA: ACM, 2021, pp. 65–77. ISBN: 978-1-4503-8383-7. DOI: 10.1145/3452296.3472888.

[249] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. "A study of network stack latency for game servers". In: *NetGames 2014*. Nagoya, Japan: IEEE, 2014, pp. 1–6. DOI: 10.1109/NetGames.2014.7008960.

[250] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[251] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: *CoRR* abs/2103.13630 (2021). arXiv: 2103.13630. URL: https://arxiv.org/abs/2103.13630.

[252] Paulius Micikevicius et al. "Mixed Precision Training". In: *ICLR 2018*. Vancouver, BC, Canada: OpenReview.net, 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[253] Mingzhen Li et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Trans Parallel Distrib Syst* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548.

[254] Inc. Rakuten Group and Rakuten Symphony. *Rakuten Launches Rakuten Symphony to accelerate adoption of cloud-native, Open RAN-based mobile networks worldwide*. Aug. 2021. URL: https://global.rakuten.com/corp/news/press/2021/0804_04.html.

[255] Kasun Indrasiri and Sriskandarajah Suhothayan. *Design Patterns for Cloud Native Applications*. 1st ed. Sebastopol: O'Reilly Media, Inc., 2021. ISBN: 978-1-4920-9071-7.

[256] Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. "Why Is It Not Solved Yet?: Challenges for Production-Ready Autoscaling". In: *ICPE '22*. Bejing: ACM, 2022, pp. 105–115. DOI: 10.1145/3489525.3511680.

[257] Milan Patel et al. *Mobile-Edge Computing – Introductory Technical White Paper*. White Paper 1. ETSI, 2014.

[258] Christoph Bachhuber, Alvaro Sanchez Martinez, Rastin Pries, Sebastian Eger, and Eckehard Steinbach. “Edge Cloud-based Augmented Reality”. In: *MMSP*. Kuala Lumpur, Malaysia, 2019, pp. 1–6. DOI: 10.1109/MMSP.2019.8901715.

[259] Patrick Kalmbach, Andreas Blenk, Wolfgang Kellerer, Rastin Pries, Michael Jarschel, and Marco Hoffmann. “GPU Accelerated Planning and Placement of Edge Clouds”. In: *NetSys*. Garching b. München, Germany, 2019, pp. 1–3. DOI: 10.1109/NetSys.2019.8854495.

[260] Marco Hoffmann et al. “SDN and NFV as Enabler for the Distributed Network Cloud”. In: *Mob. Netw. Appl.* 23.3 (June 2018), pp. 521–528. ISSN: 1572-8153. DOI: 10.1007/s11036-017-0905-y.

[261] Sahel Sahhaf et al. “Network service chaining with optimized network function embedding supporting service decompositions”. In: *Computer Networks* 93 (2015), pp. 492–505. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2015.09.035.

[262] Sameer G. Kulkarni et al. “NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains”. In: *SIGCOMM ’17*. Los Angeles, CA, USA: ACM, 2017, pp. 71–84. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098828.

[263] Robert C. Martin. *Clean Architecture - A Craftsman’s Guide to Software Structure and Design*. Prentice Hall, 2018. ISBN: 978-0-13-449416-6.

[264] György Dósa and Jiri Sgall. “First Fit bin packing: A tight analysis”. In: *STACS 2013*. Vol. 20. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2013, pp. 538–549. ISBN: 978-3-939897-50-7. DOI: 10.4230/LIPIcs.STACS.2013.538.

[265] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. “Contention-Aware Performance Prediction For Virtualized Network Functions”. In: *SIGCOMM ’20*. Virtual Event, USA: ACM, 2020, pp. 270–282. ISBN: 978-1-4503-7955-7. DOI: 10.1145/3387514.3405868.

[266] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. “Graph Attention Networks”. In: *ICLR 2018*. Vancouver, BC, Canada, 2018.

[268] Patrick Krämer, Philip Diederich, Corinna Krämer, Rastin Pries, Wolfgang Kellerer, and Andreas Blenk. “sfc2cpu: Operating a Service Function Chain Platform with Neural Combinatorial Optimization”. In: *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2021, pp. 196–205.

[269] Wei Zhang et al. "OpenNetVM: A Platform for High Performance Network Service Chains". In: *HotMIddlebox '16*. Florianopolis, Brazil: ACM, 2016, pp. 26–31. ISBN: 978-1-4503-4424-1. DOI: 10.1145/2940147.2940155.

[270] Linux Foundation. *Data Plane Development Kit (DPDK)*. 2015. URL: http://www.dpdk.org.

[271] George Prekas, Marios Kogias, and Edouard Bugnion. "ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks". In: *SOSP '17*. Shanghai, China: ACM, 2017, pp. 325–341. ISBN: 9781450350853. DOI: 10.1145/3132747.3132780.

[272] Shoumik Palkar et al. "E2: A Framework for NFV Applications". In: *SOSP '15*. Monterey, California, USA: ACM, 2015, pp. 121–136. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815423.

[273] Joao Martins et al. "ClickOS and the Art of Network Function Virtualization". In: *NSDI 14*. Seattle, WA, USA: USENIX Association, Apr. 2014, pp. 459–473. ISBN: 978-1-931971-09-6. DOI: 10.5555/2616448.2616491.

[274] The kernel development community. *CFS Scheduler*. 2020. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (visited on 06/28/2020).

[275] Robert Love. *Linux Kernel Development*. 3rd ed. Crawfordsville, Indiana, US: Pearson Education, Inc, 2010. ISBN: 978-0-672-32946-3.

[276] Jürgen Wolf. *Linux-UNIX-Programmierung*. 4th ed. Bonn, Germany: Rheinwerk, 2016. ISBN: 978-3-8362-3772-7.

[277] Linux Foundation. *3. Environment Abstraction Layer - Data Plane Development Kit 22.03.0-rc2 documentation*. URL: https://doc.dpdk.org/guides/prog%5C_guide/env%5C_abstraction%5C_layer.html%5C#known-issues (visited on 01/13/2022).

[278] *The Linux Kernel Archives*. Nov. 2022. URL: https://www.kernel.org/ (visited on 11/26/2022).

[279] Michael Kerrisk. *cgroups(7) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 01/21/2022).

[280] Tejun Heo. *Control Group v2*. 2015. URL: https://www.kernel.org/doc/Documentation/admin-guide/cgroup-v2.rst (visited on 12/01/2022).

[281] Shashank Mohan Jain. *Linux Containers and Virtualization: A Kernel Perspective*. 1st ed. Bengaluru, India: Apress Media, 2020. ISBN: 978-1-4842-6282-5.

[282] Robert Gibbons. *A primer in game theory*. Pearson Academic, 1992. ISBN: 978-0-7450-1159-2.

[283] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. New York, NY, USA: Cambridge University Press, 2007. ISBN: 978-0-521-87282-9.

[284] Sven Gronauer and Klaus Diepold. "Multi-agent deep reinforcement learning: a survey". In: *Artificial Intelligence Review* (Apr. 2021). ISSN: 1573-7462. DOI: `10.1007/s10462-021-09996-w`.

[285] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. "Neural Combinatorial Optimization with Reinforcement Learning". In: (2017). URL: `https://openreview.net/forum?id=Bk9mxlSFx`.

[286] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li. "Demystifying the Performance Interference of Co-Located Virtual Network Functions". In: *INFOCOM*. Honolulu, HI, USA: IEEE, Apr. 2018, pp. 765–773. DOI: `10.1109/INFOCOM.2018.8486246`.

[287] Zheng Peng, Wendi Feng, Arvind Narayanan, and Zhi-Li Zhang. "NFV Performance Profiling on Multi-core Servers". In: Paris, France: IEEE, June 2020, pp. 91–99. ISBN: 978-3-903176-28-7.

[288] Albert Mestres, Eduard Alarcon, and Albert Cabellos. "A machine learning-based approach for virtual network function modeling". In: *WCNCW*. Barcelona, Spain, Apr. 2018, pp. 237–242. ISBN: 978-1-5386-1154-8. DOI: `10.1109/WCNCW.2018.8369019`.

[289] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". In: *ASPLOS XV*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 129–142. ISBN: 978-1-60558-839-1. DOI: `10.1145/1736020.1736036`.

[290] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. "Thread and Memory Placement on NUMA Systems: Asymmetry Matters". In: *USENIX ATC '15*. USENIX ATC '15. Santa Clara, CA, USA: USENIX Association, 2015, pp. 277–289. ISBN: 978-1-931971-22-5. DOI: `10.5555/2813767.2813788`.

[291] J. Rao, K. Wang, X. Zhou, and C. Xu. "Optimizing virtual machine scheduling in NUMA multicore systems". In: *HPCA*. Shenzhen, China: IEEE, 2013, pp. 306–317. ISBN: 978-1-4673-5587-2. DOI: `10.1109/HPCA.2013.6522328`.

[292] Ming Liu and Tao Li. "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads". In: *ISCA*. Minneapolis, MN, USA, 2014, pp. 325–336. ISBN: 978-1-4799-4394-4. DOI: `10.1109/ISCA.2014.6853224`.

[293] Rishi Gupta and Tim Roughgarden. "Data-driven algorithm design". In: *Commun. ACM* 63.6 (2020), pp. 87–94. DOI: `10.1145/3394625`.

[294] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. “Resource Management with Deep Reinforcement Learning”. In: *HotNets ’16*. Atlanta, GA, USA: ACM, 2016, pp. 50–56. ISBN: 978-1-4503-4661-0. DOI: 10.1145/3005745.3005750.

[295] “Learning Scheduling Algorithms for Data Processing Clusters”. In: *SIGCOMM’19*. Beijing, China. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342080.

[296] Azalia Mirhoseini et al. “Device Placement Optimization with Reinforcement Learning”. In: *CoRR* abs/1706.04972 (2017). URL: http://arxiv.org/abs/1706.04972.

[297] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. “A Hierarchical Model for Device Placement”. In: *ICLR 2018*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=Hkc-TeZ0W.

[298] Azalia Mirhoseini et al. “Chip Placement with Deep Reinforcement Learning”. In: *CoRR* abs/2004.10746 (2020). URL: https://arxiv.org/abs/2004.10746.

[299] Georgios P. Katsikas, Tom Barbette, Dejan Kostiundefined, Rebecca Steinert, and Gerald Q. Maguire. “Metron: NFV Service Chains at the True Speed of the Underlying Hardware”. In: *NSDI’18*. Renton, WA, USA: USENIX Association, 2018, pp. 171–186. ISBN: 978-1-931971-43-0. DOI: 10.5555/3307441.3307457.

[300] Junaid Khalid et al. “Iron: Isolating Network-based CPU in Container Environments”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA, USA: USENIX Association, Apr. 2018, pp. 313–328. ISBN: 978-1-939133-01-4. DOI: 10.5555/3307441.3307468.

[301] Vincenzo Sciancalepore, Faqir Zarra Yousaf, and Xavier Costa-Perez. “z-TORCH: An Automated NFV Orchestration and Monitoring Solution”. In: *IEEE TNSM* 15.4 (2018), pp. 1292–1306. DOI: 10.1109/TNSM.2018.2867827.

[302] Luigi Rizzo, Paolo Valente, Giuseppe Lettieri, and Vincenzo Maffione. “PSPAT: Software packet scheduling at hardware speed”. In: *Comput. Commun.* 120 (2018), pp. 32–45. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2018.02.018.

[303] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. “RSS++: Load and State-Aware Receive Side Scaling”. In: *CoNEXT ’19*. Orlando, Florida, USA: ACM, 2019, pp. 318–333. ISBN: 978-1-4503-6998-5. DOI: 10.1145/3359989.3365412.

[304] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads". In: *NSDI 19*. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 361–378. ISBN: 978-1-931971-49-2.

[305] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency". In: *NSDI 19*. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 345–360. ISBN: 978-1-931971-49-2.

[306] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. "Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs". In: *APNet '19*. Beijing, China: ACM, 2019, pp. 71–77. ISBN: 978-1-4503-7635-8. DOI: `10.1145/3343180.3343184`.

[307] *Amtsblatt 23*. Tech. rep. Bonn, Germany: Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen, Aug. 2021, pp. 1575–1633.

[308] Rishi Gupta and Tim Roughgarden. "Data-Driven Algorithm Design". In: *Commun. ACM* 63.6 (May 2020), pp. 87–94. ISSN: 0001-0782. DOI: `10.1145/3394625`.

[309] Center for Applied Internet Data Analysis (CAIDA). *Trace Statistics for CAIDA Passive OC48 and OC192 Traces*. 2020. URL: `https://www.caida.org/data/passive/trace%5C_stats/` (visited on 06/28/2020).

[310] Amy Greenwald, Jiacui Li, and Eric Sodomka. "Solving for Best Responses and Equilibria in Extensive-Form Games with Reinforcement Learning Methods". In: *Rohit Parikh on Logic, Language and Society*. Basel, Swiss: Springer, Cham, 2017, pp. 185–226. ISBN: 978-3-319-47843-2. DOI: `10.1007/978-3-319-47843-2\_11`.

[311] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *PMLR*. Vol. 15. PMLR. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323.

[312] Bowen Baker et al. "Emergent Tool Use From Multi-Agent Autocurricula". In: *arXiv e-prints* (Sept. 2019), arXiv:1909.07528.

[313] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: `https://openreview.net/forum?id=YicbFdNTTy`.

[314] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). URL: `http://arxiv.org/abs/1509.06461`.

[315] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized Experience Replay". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: http://arxiv.org/abs/1511.05952.

[316] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). URL: http://arxiv.org/abs/1511.06581.

[317] Eric Liang et al. "Ray RLLib: A Composable and Scalable Reinforcement Learning Library". In: *CoRR* abs/1712.09381 (2017). arXiv: 1712.09381. URL: http://arxiv.org/abs/1712.09381.

[318] Max Jaderberg et al. "Population Based Training of Neural Networks". In: *CoRR* abs/1711.09846 (2017). arXiv: 1711.09846. URL: http://arxiv.org/abs/1711.09846.

[319] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Trans. Neural Networks Learn. Syst.* 32.1 (2021), pp. 4–24. DOI: 10.1109/TNNLS.2020.2978386.

[320] Nikolas Adaloglou. "Intuitive Explanation of Skip Connections in Deep Learning". In: *https://theaisummer.com/* (2020). URL: https://theaisummer.com/skip-connections/.

[321] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *CoRR* abs/1607.06450 (2016). arXiv: 1607.06450. URL: http://arxiv.org/abs/1607.06450.

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

[323] I. H. Witten and E. Frank. *Data mining, practical machine learning tools and techniques*. San Francisco: Morgan Kaufmann Publishers, 2011.