UNIVERSIDADE FEDERAL DO PAMPA

Francisco Germano Vogt

Towards In-Network Neural Networks

Alegrete
2021

Francisco Germano Vogt

Towards In-Network Neural Networks

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Supervisor: Prof. Dr. Marcelo Caggiani Luizelli

Alegrete
2021

SERVIÇO PÚBLICO FEDERAL

MINISTÉRIO DA EDUCAÇÃO

Universidade Federal do Pampa

FRANCISCO GERMANO VOGT

Towards In-Network Neural Networks

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação

Monografia defendida e aprovada em: 12, maio de 2021.

Banca examinadora:

_____

Prof. Dr. Marcelo Caggiani Luizelli

Orientador

UNIPAMPA

_____

Prof. Dr. Arthur Francisco Lorenzon

UNIPAMPA

Prof. Dr. Fábio Diniz Rossi

IFFar

Assinado eletronicamente por **ARTHUR FRANCISCO LORENZON**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 10:28, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Fábio Diniz Rossi**, **Usuário Externo**, em 12/05/2021, às 10:39, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **MARCELO CAGGIANI LUIZELLI**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 14:23, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0523612** e o código CRC **324330AF**.

This work is dedicated to my family, friends and everyone who somehow participated in my academic trajectory, but especially my mother, "Dona Noemi", who raised me alone and went to great lengths to get me here.

Acknowledgements

First of all, I would like to thank my family, especially my mother Noemi, my uncle Lucio and my aunt Izolde, for the love, affection, understanding, and help they have given me so far. Certainly, without your support, none of this would have been possible.

In addition to the family, I would like to especially thank two professors, Dr. Marcelo Caggiani Luizelli, a friend and advisor, whom I am very proud to have been guided. So, thank you for all the motivation, teachings, opportunities, and friendship offered throughout my graduation. Also, I'd like to give a special thanks to Dr. Arthur Francisco Lorenzon for his sincere friendship. He was always willing to help, advise, guide, and draw attention to when necessary.

Concerning my friends, that fortunately, were many, I would like to especially thank Filipo and Rafael, the first two colleagues and friends I met on the first day of school and that participated from the beginning to the end of my academic and private life, sharing unforgettable moments I will keep with me with great affection.

In addition to them, Ariel, a more than special colleague and friend with whom I worked the most, studied, learned, taught, and who certainly played a fundamental role in my trajectory.

And of course, I can't forget the great friends I made not only inside but also outside from college, and with whom I had the immense pleasure of sharing these four years. Among them, are Sérgio, Lauriano, Douglas, Cassiano, Igor, Takeshi, and others. I may have forgotten to mention some people. However, they know they were very important to me.

"I've always been a dreamer, that's what keeps me alive" (Racionais MC's)

Resumo

Nos últimos anos as técnicas de machine learning (ML) têm se mostrado eficientes quando aplicadas a problemas de operação e gerenciamento de rede, existindo vários estudos recentes que buscam aplicar essas técnicas em áreas e tecnologias de rede especificas. Nesta monografia, dá-se o primeiro passo para uma implementação de plano de dados inteligente empregando Redes Neurais In-Network. O problema consiste em mapear um conjunto de redes neurais artificiais (RNAs) em uma infraestrutura de rede programável, seguindo uma série de restrições (como por exemplo mapear apenas um único neuronio por dispositivo de rede) que buscam otimizar o funcionamento da RNA na rede. Para resolver este problema, inicialmente é formalizado um problema de otimização, utilizando o modelo MILP *(Mixed-Integer Linear Programming)*. Em seguida, desenvolve-se uma meta-heuristica baseada em um algoritmo construtivo e, outros dois algoritmos, sendo um guloso e outro aleatório. Ambos buscam encontrar uma solução válida para o problema com uma mínima quantidade de recursos utilizados (por exemplo, memória e processamento). O objetivo então é avaliar as soluções encontradas pelas estratégias propostas, em comparação a uma redução do problema para uma instância do VNE, problema similar ao que está sendo resolvido, porém com restrições mais simples. Além disso, busca-se avaliar o impacto de alguns parâmetros e métricas, como número de fluxos disponíveis/utilizados, na qualidade das soluções geradas. Os resultados mostram que as técnicas de meta-heurística e VNE não são capazes de encontrar soluções de maneira escalavél (em questões de processamento e memória) para o problema, limitando-se a instâncias pequenas do problema. Por outro lado, os algoritmos guloso e aleatório conseguem mapear o número máximo de RNAs possível, considerando RNAs de 5 neurônios e uma topologia do tipo fat-tree de 20 dispositivos, já para a topologia de 80 dispositivos mapeamos 87.5% das RNAs possíveis. Além disso, os algoritmos são capazes de mapear essas ANNs com uma número baixo de fluxos de rede disponível, encontrando por exemplo o número máximo de mapeamentos possível com apenas 20 fluxos disponíveis, para a topologia de 20 dispositivos.

Palavras-chave: Software-Defined Networks(SDN). Machine Learning (ML)

Abstract

In recent years, machine learning (ML) techniques have been shown to be efficient when applied to network operation and management problems, and there are several recent studies that seek to apply these techniques in specific network areas and technologies. In this work, the first step is taken towards an intelligent data plan implementation using In-Network Neural Networks. The problem consists in mapping a set of Artificial Neural Networks Artificial Neural Networks (ANN) in a programmable network infrastructure, considering a series of restrictions (e.g., mapping only a single neuron per network device) that seek to optimize the operation of the ANNs in the network. To solve this problem, an optimization problem is initially formalized, using the Mixed-Integer Linear Programming (MILP) model. To try to find feasible solutions to the problem, we developed a math-heuristic based on solutions generated by a constructive heuristic, besides two other algorithms with polynomial complexity, based on random and greedy decisions. Both seek to find a valid solution to the problem with a minimum amount of resources used (for example, memory and processing). The objective is to evaluate the solutions found by the proposed strategies, in comparison to a reduction of the problem for an instance of the VNE, a problem similar to the one being solved, but with simpler restrictions. In addition, it seeks to assess the impact of some parameters and metrics, such as the number of available / used flows, on the quality of the solutions generated. The results show that the math-heuristic and VNE techniques can not generate scalable solutions (in terms of processing and memory) to the problem, being able to solve only small instances of the problem. On the other hand, the greedy and random algorithms can map the maximum number of ANNs possible, considering ANNs with 5 neurons and a fat-tree topology of 20 devices. Similarly, in the topology of 80 network devices, we map 87.5 % of the possible ANNs. In addition, the algorithms can map these ANNs with a low number of available network streams, finding, for example, the maximum number of mappings possible with only 20 flows available for the topology of 20 devices.

Key-words: Software-Defined Networks(SDN). Machine Learning (ML).

List of Figures

List of Tables

Lista de siglas

AMD Advanced Micro Devices

ANN Artificial Neural Networks

API Application Programming Interface

ARP Address Resolution Protocol

ASIC Application-Specific Integrated Circuit

CLI Command-Line Interface

DDoS Distributed Denial of Service

DNN Deep Neural Network

FPGA Field Programmable Gate Array

IBM International Business Machines

ICMP Internet Control Message Protocol

IN^3-P In-Network Neural Network Problem

INT In-Band Network Telemetry

MILP Mixed-Integer Linear Programming

MLP Multilayer Perceptron

NIC Network Interface Card

NOS Network Operating System

NVGRE Network Virtualization using Generic Routing Encapsulation

OUI Organizationally Unique Identifier

P4 Programming Protocol-independent Packet Processors

QoE Quality of Experience

RAM Random Access Memory

SDDCN Software-Defined Data Center Networks

SDN Software-Defined Network

SNAP Sub-Network Acess Protocol

SNMP Simple Network Management Protocol

TCP Transmission Control Protocol

ToS Type of Service

UDP User Datagram Protocol

VNE Virtual Network Embedding

VxLAN Virtual Extensible LAN

Contents

1 Introduction

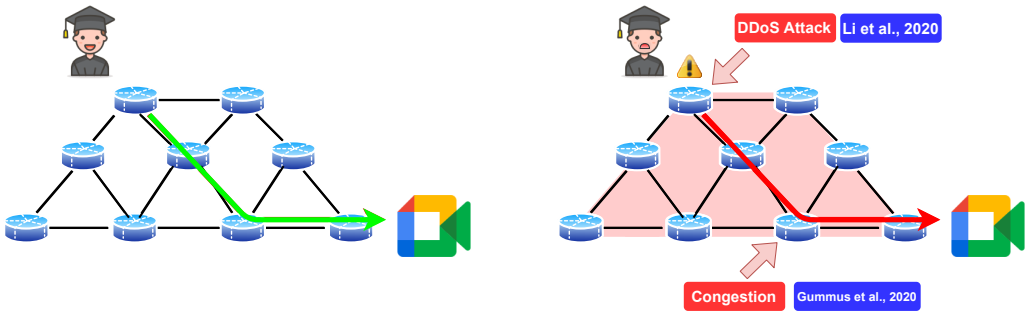
In this chapter, we discuss the problem of operating in-network neural networks. Initially, we introduce a motivation for the use of in-network neural networks in current programmable networks. Then, we outline the research problem and the main contributions of this final work.

1.1 Context and Motivation

Data plane programmability is currently reshaping traditional network device usage and operation, and ultimately the network infrastructure as a whole. Modern academia and industry efforts have shown a multitude of disruptive network applications such as fine-grained network telemetry (HOHEMBERGER et al., 2019), in-network caching and data aggregation (XIONG; ZILBERMAN, 2019a), and adaptive routing mechanisms (Pizzutti; Schaeffer-Filho, 2019) – to name a few examples. The rapid adoption of programmable data plane is due to its flexibility offered by high-level network programming languages (e.g., Programming Protocol-independent Packet Processors (P4) (BOSSHART et al., 2014a)) that enables an easy way to adapt custom-made functions to data planes (e.g., (BASAT et al., 2017)).

Despite the ongoing shift from control plane decisions to data plane-based ones, we are still far from intelligent network devices, for example, capable of executing more complex algorithms, such as neural networks. Most decisions taken by the data plane rely on (i) pre-computed lookup tables (e.g., routing), and (ii) specific network states or conditions (e.g., network congestion event), making data plane decisions mostly deterministic and dependent on the control plane algorithms. We strongly believe that we can break this control-loop dependency and allow data planes to learn by themselves the network states and make appropriate choices. It would bring the benefits of real-time decisions at no extra cost related to communication with the control plane.

As an example, suppose that someone is presenting an academic work using an



(a) Initial/Normal network state (b) Abnormal network state

Figure 1 – Example of networking problems that can be identified with in-network neural network.

online platform. At the beginning, everything is going as expected, that is, there is no problem with the network connection (e.g., video/audio freezing). However, from a given point on, the network connection starts to be unstable (e.g., high packet losses or low throughput), decreasing the Quality of Experience (QoE). These network problems might have numerous root causes, such as a Distributed Denial of Service (DDoS) attack or packet congestion on a given forwarding device (see Figure 1b). In the given example, we assume that the root cause of the observed networking performance degradation has been mainly caused by a DDoS attack, because is a common place in today's computer networks (FOULADI; SEIFPOOR; ANARIM, 2013). By using a traditional mitigation approach to prevent/stop such attacks, data planes would send network statistics (or sampled network traffic packets) to a monitoring application that eventually would notify the control plane to divert part of network traffic. The required time to identify the anomaly, it notify the control plane, and take an action is usually in the orders of seconds to minutes. Back to the initial example, the presenter would have to wait (and his/her audience too) until the network problem is solved to properly resume the presentation. Observe that if this could be performed entirely by the data plane, we would potentially reduce the time to block undesirable network traffic (in the orders o milliseconds) at the cost of sending no extra packets to monitoring applications.

To provide an intelligent data plane, we give the first step towards an efficient implementation of in-network neural networks in programmable devices. As far as we know, little has yet been done about in-network neural networks. Xiong and Zilberman (XIONG; ZILBERMAN, 2019a) have focused on in-network classification with non-neural network machine learning approaches (e.g., clustering). N2Net (SIRACUSANO; BIFULCO, 2018) and BaNaNa (SANVITO; BIFULCO, 2018) have introduced the first implementation of binary neural networks on programmable devices. Unlike these efforts, in this work, we aim to optimize how multiple neural networks are mapped onto programmable data planes. Each neural network might have different requirements (e.g., inter-layer communications) and be trained for a particular purpose (e.g., packet classification). By running neural networks directly on the data planes, we can benefit from high-performance forwarding devices to (i) learn network behaviors while packets are forwarded – i.e., without the overhead of mirroring network packets to outside analyzers; (ii) take decisions at real-time – i.e., data plane latency is in the order of nanoseconds, while traditional machine learning applications operate at the millisecond scale; (iii) reduce the need to continually update key-values on lookup tables as the decisions rely on the learned behavior.

1.2 Objectives and Contributions

In this work, we first focus on formally defining the In-Network Neural Network Problem and optimally model it as a specialization of the well-studied multi-commodity

flow problem. The idea consists of optimally embedding and running multiple specialized neural networks into programmable data planes. Neural networks are composed of layers of neurons that communicate among them. Neurons can run independently on data planes. Its interconnections take advantage of In-Band Network Telemetry (INT) capabilities – i.e., they leverage active network flows to encapsulate and route inter-layer communications – reducing overheads in the physical networks (e.g., extra packets). The main contributions of this work are as follows:

- The formalization of the In-Network Neural Network by means of an optimal Integer-Linear Programming model;
- The design of efficient and scalable algorithmic methods to compute quality-wise solutions promptly.

1.3 Outline

The remainder of this thesis is organized as follows. In Chapter 2, we overview the networking/machine-learning background required to understand this work. Further, we discuss the main research efforts regarding in-network data plane computation. In Chapter 3, we introduce the in-network neural network problem, presenting a description and formalization of the problem. Then, we introduce the designed strategies to efficiently solve the problem. In Chapter 4, we present the methodology used for evaluation and the main results. Finally, Chapter 5 presents the final remarks and future work.

2 Background and Related Work

In this chapter, we overview network programmability from the control and data plane perspective. Then, we review machine learning techniques and their applications to the computer networking domain. Last, we review recent research efforts towards applying machine learning to networking problems.

2.1 Network Programmability

2.1.1 Software-Defined Networking

Since the adoption of traditional IP networks, several challenges were faced, and still many hamper the evolution of current networks. For example, the transition between IPv4 and IPv6 started more than two-decade ago (WU et al., 2012), but it is still largely incomplete, despite being a protocol update. Situations similar to this happen because traditional IP networks are complex and very hard to manage (BENSON; AKELLA; MALTZ, 2009). On top of that, configuring current networks according to predefined policies is complicated, which turns the network inflexible and ineffective in reaction to faults and workload changes. In case a network operator needs to set up a new set of high-level policies in the network, he/she needs to configure each network device separately. Besides, the network operator would implement these policies using low-level commands, device vendor-specific Command-Line Interface (CLI) (Kim; Feamster, 2013). Another factor is the network devices are vertically integrated, meaning that the management, the forwarding, and the control planes are tightly coupled. The aforementioned configuration complexity turns the network hard (or even infeasible) to work dynamically, making most of the decisions "plastered" and non-adaptive to faults and network changes. Therefore, enforcing the required policies in a dynamic environment – such as on the Internet – is highly challenging, considering that automatic and dynamic reconfiguration is unavailable in current networks.

In traditional network architectures – as aforementioned – the management, the data, and the control planes are tightly coupled. The control plane is responsible for running protocols (e.g., routing) and populate forwarding tables that the data plane implements. In turn, the data plane is responsible for forwarding network packets according to the rules (or policies) defined by the control plane. In addition to that, the network operation splits into a third plane – also known as the management plane. Software services run in the management in order to properly configure the control plane algorithms (e.g., to set up parameters of a given routing protocol). To summarize, the management plan defines the policies, the control plane enforces those policies, and the data plane forwards the data according to the applied ones. Figure 2 shows an abstraction of the interplay among these three planes.

Software-Defined Network (SDN) is a network paradigm that has emerged as an

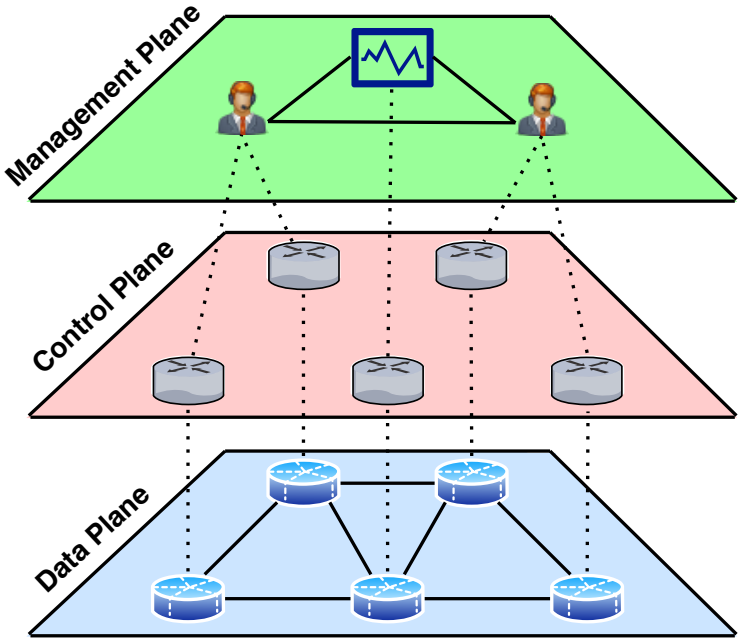


Figure 2 – Division of networking funcionaliy

alternative for decoupling the control and data planes of networking devices. By decoupling the control plane and the data plane, SDN turns network devices into simple forwarding devices (also known as white boxes) – that is, network devices become simple packet forwarders, implementing only a simple data plane. Then, the complexity of the control plane is moved from network devices to a logically centralized entity, known as SDN controller or Network Operating System (NOS). The NOS is an entity responsible for enforcing network policies and managing multiple control network applications. There are many benefits of decoupling these two planes. First, network management is eased since it is possible to establish a set of network policies independently and in a centralized manner. Second, it fosters the design and development of new control plane applications as applications become architecture-independent. Third, it reduces the cost of networking devices as the Application-Specific Integrated Circuit (ASIC) chips is considerably simpler.

The communications between the control plane and the data plane is performed through a well-defined Application Programming Interface (API) between networking devices (e.g., switches) and the SDN controller. By using this interface, the SDN controller has control over the state of the data plane, and it can (re-)configure the applied policies dynamically. An example of such API is the OpenFlow(MCKEOWN et al., 2008; ONF, 2014) protocol. OpenFlows has became the standard and de-facto protocol used by commercial SDN controllers and networking devices.

An OpenFlow switch[1] is a network device that implements the OpenFlow protocol.

[1] We use OpenFlow switch to refer to any OpenFlow-enabled forwarding device.

The OpenFlow switch has one or more forwarding tables. These tables have a set of packet-handling rules, and each rule matches a subset of the network traffic. These rules can run a set of actions in the network packets (e.g., forward/drop packets, or modify a specific packet header), and depending on the rules installed by the SDN controller, the OpenFlow switch behaves differently. For instance, a forwarding device may act as a firewall if the control plane installs rules into the data planes to block/allow the packet forwarding of specific IP prefixes. Other examples include network load balancing, traffic shaping, and routing – to name a few. Currently, the majority of networking vendors included in their devices support to the OpenFlow protocol. Table 1 shows the set of match fields supported in OpenFlow 1.0 (FUNDATION, 2009).

Table 1 – Summary of OpenFlow fields (OF version 1.0).

Field	**Overview/Notes**	**Size (bits)**
Ingress Port	Numerical representation of incoming port, starting at 1	(Implementation dependent)
Ethernet source address		48
Ethernet destination address		48
Ethernet type	An OpenFlow switch is required to match the type in both standard Ethernet and 802.2 with a Sub-Network Acess Protocol (SNAP) header and Organizationally Unique Identifier (OUI) of 0x000000. The special value of oxo5FF is used tomatch all 802.03 packets without SNAP headers	16
VLAN id		12
VLAN priority	VLAN PCP field	3
IP source address	It can be subnet masked	32
IP destination address	It can be subnet masked	32
IP protocol	It only use the lower 8 bits of the Address Resolution Protocol (ARP) op-code are used	8
IP Type of Service (ToS) bits	Specify as 8-bit value and place ToS in upper 6 bits	6
Transport source port / ARP Type	It only utilized the lower 8 bits used for Internet Control Message Protocol (ICMP) type	16
Transport destination port / ICMP Code	It only utilized the lower 8 bits used for ICMP code	16

An important benefit of the SDN architecture is the centralization of the control plane. The SDN controller is logically centralized, which means it may be physically

distributed – as is the case for (KOPONEN et al., 2010) and (JAIN et al., 2013). The logical centralization of SDN controllers offer many benefits to the network, for example, easing the changing process of network policies by using high-level languages (such as Pyretic (REICH et al., 2013) and Frenetic (FOSTER et al., 2010). Moreover, as the controller has global knowledge of the network state, it becomes simpler to develop robust applications/services for the network management. Figure 3 illustrates a side-by-side comparison between the operation of network functions in traditional networks and the SDN architecture. Observe in the figure that in traditional networks, the control plane and the data plan are bundled together within the networking devices, and the network functions (e.g., firewall and load balancer) are implemented using physical middleboxes. In turn, in SDN-based networks, we observe that control plane applications are centralized in the NOS, while network functions might be run as a control plane application (MIJUMBI et al., 2015).

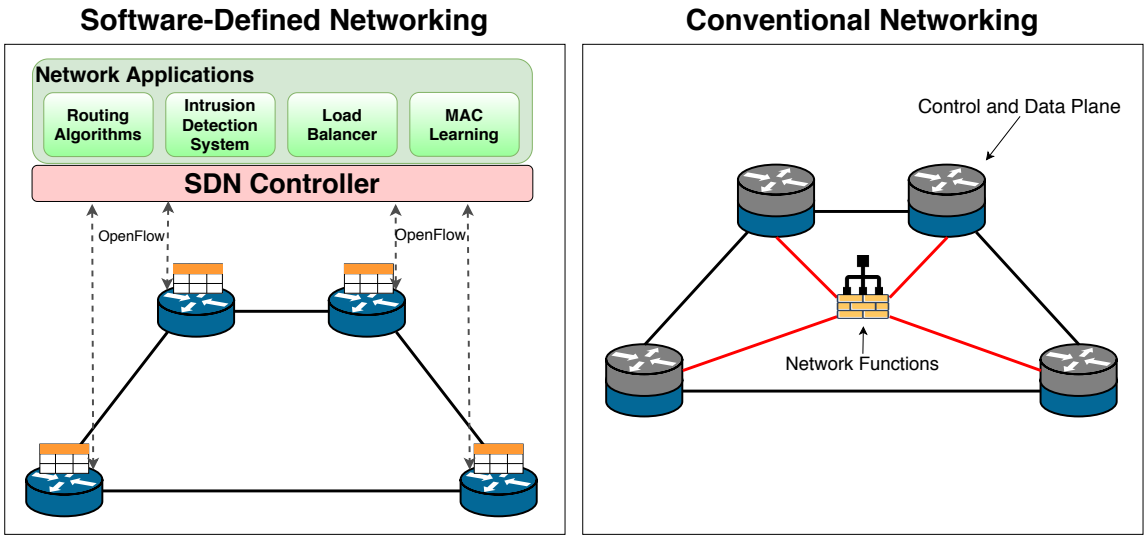


Figure 3 – Comparing traditional networks with SDN-based ones. Figure adapted from (KREUTZ et al., 2015).

2.1.2 Programmable Data Planes

As previously discussed, the decoupling between the control plane and data plane introduced by the SDN architecture brought up the possibility to program the data plane with high-level languages (e.g., Frenetic (FOSTER et al., 2010)). This approach has been mainly possible due to the establishment of a vendor-agnostic interface such as OpenFlow.

OpenFlow (ONF, 2014) was designed to be a simple and pragmatic vendor-agnostic interface. In its early ages, it had a very limited set of 12 header fields for matching/action. However, as time goes by, the requirements for more specific header implementations become a requirement and, consequently, the number of header and actions available

increased significantly. For instance, in 2013, that number surpassed 41. Such increase in the number of headers and actions supported by the OpenFlow specification has brought the benefit of expressing control planes programs to a wider variety of protocols (e.g., non-IP protocols or new encapsulation protocols such as Network Virtualization using Generic Routing Encapsulation (NVGRE) (GARG; WANG, 2015)/Virtual Extensible LAN (VxLAN) (MAHALINGAM et al., 2014)). However, there has also been a side effect. The data plane implementation became complex and directly dependent on the OpenFlow specification.

As a consequence, the life-cycle of more complex control plane programs started to take more time as it has a strongly dependency of the OpenFlow specification and data plane implementation. For instance, suppose one control plane application needs to handle a new transport protocol (i.e., different from conventional Transmission Control Protocol (TCP)/User Datagram Protocol (UDP)). That would require implementing it directly on the data plane (i.e., in ASIC or software data plane) and update the OpenFlow specification. As one can imagine, that would require at least a few months (if not years) to become a reality in commodity switches.

These limitations of OpenFlow motivated the design of a high-level language for network data plane programming. P4 (BOSSHART et al., 2014b) has been designed to ease the burden of data plane programmability and to increase the flexibility and expressiveness of control plane programs. In short, P4 is a high-level language that defines the behavior of data planes. P4 works with SDN interfaces (e.g., OpenFlow or P4Runtime (CONSORTIUM et al., 2017)). However, these interfaces work as a communication medium between control and data planes, rather than explicitly dictating the protocol/headers/actions specification.

By using P4 language, a networking programmer can easily specify how the packets are supposed to be processed in the forwarding device (e.g., switches or Network Interface Card (NIC)s). The written code is then compiled to generate low-level instruction to a variety of networking targets (e.g., Field Programmable Gate Array (FPGA)s, SmartNICs, software switches, etc). The P4 forwarding model works slightly differently when compared to the OpenFlow data plane. In P4, switches forward incoming packets using multiple stages of match/action. Moreover, while in OpenFlow the matches are realized using a fixed parsing (with a limited set of header fields), P4 supports a programmable parser to allow new headers/protocols.

Yet, the P4 forwarding model can be described in two types of operations: (i) configure and (ii) populate. These operations occur in two distinct phases. First, the configure operation program (or load) the parser and organize the order of match/action stages, specifying which header fields are processed in each stage. Second, the populate operation manages (adding and/or removing) table entries, according to the parser, inserting in the match/action tables specified during configuration.

Figure 4 illustrates an abstraction of the P4 forwarding model. When a packet is received on the switch, the first stage is the parser. In the parser, the packet headers and payload are extracted and stored in internal memory structures. The packet payload is buffered (unavailable for matching), and the header fields are used to match with the protocols supported by the switch and define the sequence of actions to be performed. Then, the extracted headers are forward to match/action tables. The match/action tables are divided into ingress and egress, although both can modify the headers fields, they do different tasks. Ingress match/action tables are responsible for determining the egress port/queue into which the packet is placed. Then, based on this decision, the packet is forwarded, replicated, or dropped. In the egress match/action tables are performed only per-instance modifications as an example by adding or removing monitoring information (e.g., timestamp). Last, the packet is remounted on the departure, finishing processing.

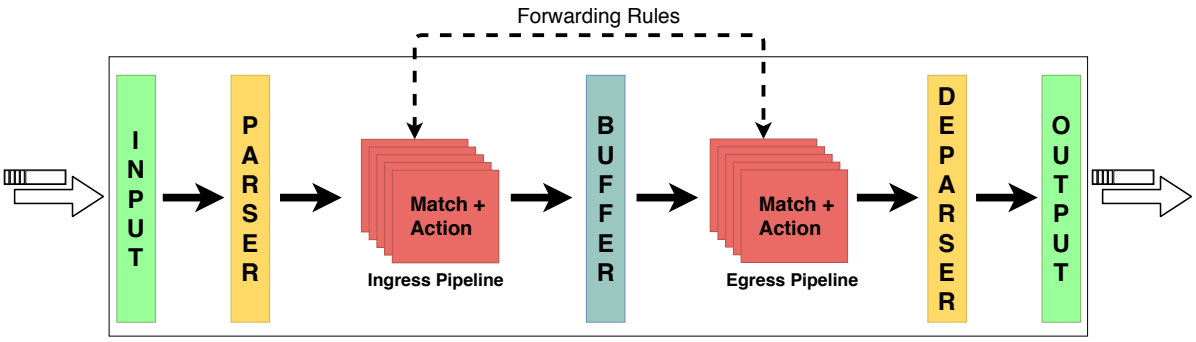


Figure 4 – P4 Pipeline

2.1.2.1 Data Plane Application: In-band Network Telemetry

Network programmability enables a multitude of benefits related to the management and operation of network infrastructures. As previously discussed, control plane programmability enabled to design of logically centralized algorithms with global network knowledge (e.g., monitoring mechanism). On top of that, the flexibility offered by high-level data plane programming languages adds up more flexibility to specify data plane programs according to the needs of control plane programs. These two programmability levels allow the design of new networking applications for a variety of purposes – for instance, identification of heavy-hitters, TCP in-cast or traffic imbalance, DDoS attacks mitigation and identification of anomalies, and novel network monitoring mechanisms. For this thesis, we focus on INT – an emerging network monitoring mechanism – that provides higher network-wide visibility to network operators (LIU et al., 2018). Next, we describe it in detail.

In-band network telemetry enables the collection of network statistics with higher granularity when compared with current protocols (e.g., Simple Network Management Protocol (SNMP) (CASE M. FEDOR, 1989)). INT can be realized in two ways: (i)

out-band – using active probes (PAN et al., 2019), and (ii) in-band – using packets of active network flow to encapsulate network statistics (HOHEMBERGER et al., 2019). In the former, probe packets are created specifically to contain telemetry information and contain specific instructions about what telemetry information should collect on what devices. In the latter, the telemetry information is incorporated into the user's packages, using for example header fields or the payload itself.

Figure 5 illustrates the process of in-band network telemetry. First, the packet is received by a programmable network device. Through the analysis of the telemetry instruction that the package carries (for example which data to collect and where / when) the network device incorporates the telemetry information (e.g., delay in the queue, timestamp) in, for example, some header field, and finally, the packet is forwarded by the network device containing the information, which will be extracted and sent to an information collector later, before the package is delivered.

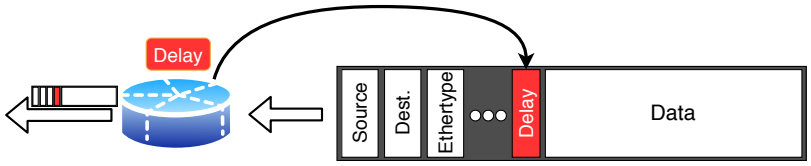


Figure 5 – Embedding telemetry information in the packet

In the telemetry process, a wide range of information can be collected (e.g., queue occupancy, data plane processing time, switchID). This information can be used by monitoring applications for the management of network infrastructures.

2.2 Machine Learning for Networking

Machine learning consists of techniques that enable a system to acquire knowledge through the analysis of data sets(MITCHELL et al., 1997). More specifically, machine learning can identify hidden patterns in data, by means of an appropriate training technique. The patterns learned during the training phase are then used to analyze unknown data. Recent advances in machine learning make it possible to find complex patterns in the data, allowing these techniques to be efficiently used in decision-making real-world problems. For instance, machine learning techniques have been used in several emergent areas such as autonomous vehicles, health care, computational problems, and more recently in the networking domain.

The emergence and consolidation of network programmability have enabled machine learning techniques to be seamlessly applied in the networking domain (BOUTABA et al., 2018). There are many recent networking applications (both in the control and data plane) that rely on machine learning algorithms and models. For instance, traffic classification, congestion control, and network security – to mention a few. This is pos-

sible mainly because of OpenFlow/P4Runtime API. These APIs have enabled a flexible inter-communication between the control and data plane. Employing that, it is possible to write both control and data plane applications that can collect input data that is later used by machine learning algorithms/models.

Next, we describe two learning paradigms: (i) supervised learning, and (ii) unsupervised learning. Further, we dive into its applications in networking operations, focusing mainly on artificial neural networks.

2.2.1 Supervised Learning

Supervised learning is the area of machine learning where most problems are already well defined, and where exist majority of successful applications. The key point of supervised learning is the fact that the data used for training contains the desired answer. This means that when using supervised learning, we try to predict dependent variables based on a list of independent variables. For example, we can try to predict the delay of a packet based on the distance between source and destination, the bandwidth available, and the network state, where the delay is the dependent variable, and the distance, bandwidth, and network state are the independent variables.

To perform the learning (i.e., acquire the ability of predicting the dependent variables), supervised techniques utilize labeled training datasets (i.e., data with the answers or classes to be predicted), divided between training and test. This approach of learning usually is used in classification and regression problems, which have different characteristics and goals that should be taken into account when building the model. Classification problems aim to predict discrete classes or categories. For example, suppose a given algorithm has already learned to classify packets according to a set of classes of applications (e.g., HTTP, FTP, or DNS). Then, when a packet of one of these classes comes into the network, the networking devices would have the ability to classify it as expected. In regression problems, the prediction considers continuous value. For example, in network traffic prediction, one possible alternative would be to establish a relationship between previously observed traffic to predict future traffic volume. Examples of regression algorithms include linear regression, multiple linear regression, and non-linear regression.

As the focus of the work is artificial neural networks, we discuss it in more detail, based in (BRAGA, 2000). Artificial neural networks is a technique of computational learning that shows a mathematical model based human brain. These models are based on simple processing units called neurons. A neural network can contain thousands (or even hundreds of thousands) neurons, while the human brain can contain up to billions. Neurons process only local data and interconnect with other neurons (when necessary) through a synapses process, that is used to propagate the knowledge acquired to other neurons.

Figure 6 illustrates the processing of the simplest neural network – named Percep-

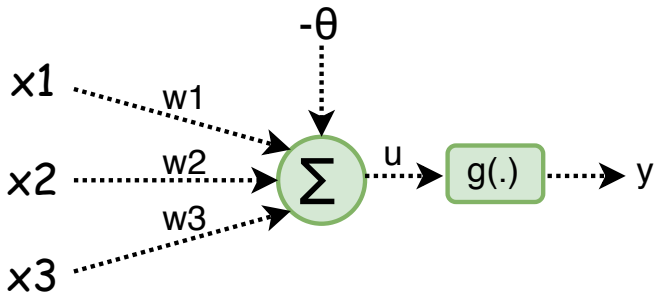


Figure 6 – Example of Peceptron

tron – that has only one neuron in its structure. The perceptron receives as input a set of variables (x_1, x_2, ..., x_n) that represents the data used to predict a given class/pattern. The neuron realizes a weighted summation (Equation 2.1) among the input data and its corresponding weight (w_1, w_2, ..., w_n). Each assigned weight represent the importance of a give synapse. The result of this summation is then used as input of an activation function $g(.)$ (Equation 2.2).

$$u = \sum_{i=1}^{n} Xi * Wi - \theta \tag{2.1}$$

$$y = g(u) \tag{2.2}$$

The activation function aims to limit the output amplitude of the neuron, that is, to normalize the value within a closed range, for instance $[0, 1]$. There are several activate functions such as Heaviside and identity (linear functions), and Sigmoid, Gaussian, and Hyperbolic Tangent (non-linear functions). The Heaviside function, which is usually used by the Perceptron, is described in the Equation 2.3:

$$y = \begin{cases} 1 & \text{if } g(u) >= 0 \\ 0 & \text{if } g(u) < 0 \end{cases} \tag{2.3}$$

Furthermore, during the supervised training, the weights of the synapses are updated, following the rule of Hebb, described in Equation 2.4. In the equation, η represents the learning rate and d^{k} represents the desired value for sample k. Note that if the value of d^{k} is equals to y (the result equals desired), the weights would not be modified.

$$Wi^{\,current} = Wi^{\,previous} + \eta * (d^{(k)} - y) * Xi^{(k)} \tag{2.4}$$

In more sophisticated neural networks, the main evolution is the existence of multiples neurons, organized in layers. The artificial Multilayer Perceptron (MLP) is an example of this type of network, where the output of a neuron is used how the input other neurons of the following layers. This type of network works in feedforward mode

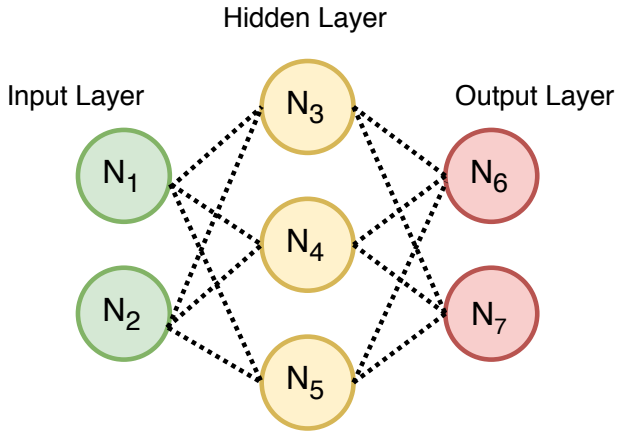


Figure 7 – Example of a Artificial Multilayer Perceptron (MLP).

allowing it to deal with nonlinearly separable problems. Figure 7 shows an example of an artificial multilayer network, composed of three layers, two neurons in the input layer, three in the intermediate layer, and two in the output layer.

In multi-layer networks, the training occurs differently, by using an algorithm named backpropagation. Initially, the algorithm calculates the error through Equation 2.5, where rr represents the expected value and ro the value obtained by the neuron.

$$e_{\mathrm{j}} = ro_{\mathrm{j}} - rr_{\mathrm{j}} \tag{2.5}$$

Then, for updating the weights and propagate corrections, it is necessary to calculate the local gradient. For the neurons of the output layer, the local gradient is the result of Equation 2.5 (i.e., the error) multiplied by the derivative of its activation function, represented by $\varphi(\mathrm{v})$ – described in Equation 2.6.

$$\delta_{\mathrm{j}} = \varphi_{\mathrm{j}}(v_{\mathrm{j}}) * e_{\mathrm{j}} \tag{2.6}$$

Then, for the neurons in the remaining layers, Equation 2.7 is used to calculate the local gradient. In this equation, the derivative of its activation function, represented by $\varphi(\mathrm{v})$, is multiplied by the summation of another local gradient, symbolized by δ, obtained by the retro propagation from previous neurons (in this case, neuron k * synaptic weights between neurons k and j).

$$\delta_{\mathrm{j}} = \varphi_{\mathrm{j}}(v_{\mathrm{j}}) * \sum_{k} \delta_{\mathrm{k}} * W_{\mathrm{k\,j}} \tag{2.7}$$

After calculating the local gradient, we can update the weights from Equation 2.8, where η is the rate learning.

$$\Delta W_{\mathrm{ji}} = \eta * \delta_{\mathrm{j}} * y_{\mathrm{i}} \tag{2.8}$$

In the context of computer networks, ANNs are used to solve several problems, such as traffic prediction (ZHU; ZHANG; QIU, 2013; LI et al., 2016), traffic classification (AULD; MOORE; GULL, 2007) and (SUN et al., 2010), congestion control (GEURTS; KHAYAT; LEDUC, 2004; HARIRI; SADATI, 2007), and network security (PAN et al., 2003; MORADI; ZULKERNINE, 2004).

2.2.2 Unsupervised Learning

Supervised learning cannot solve any learning problem. In some cases, having access to a labeled dataset might be infeasible. Besides, there are problems that the goal is to find possible unknown relationships among the available data – without knowing in advance what might be these relations. For those cases, unsupervised learning is recommended and has been applied in a multitude of applications – for instance, anomaly detection, recommendation systems, and data visualization.

Some many algorithms and models are applied to unsupervised learning. The most commons are clustering, association rule learning, and dimension reduction. The clustering seeks to find a similarity between available data and group them into specific clusters. The main algorithms are K-means(LIKAS; VLASSIS; VERBEEK, 2003), Single Linkage(GOWER; ROSS, 1969), DBSCAN (SCHUBERT et al., 2017), Mean-Shift(CHENG, 1995), and Complete Linkage(DEFAYS, 1977). In turn, the association rule learning is commonly used when there is a sequence of incoming data and one wants to find patterns. Traffic prediction is an example where packets are online received. The goal, in this case, consists of finding a pattern to eventually predict future events. Some techniques used in association rules are Sequence, Apriori(BORGELT; KRUSE, 2002), and Carma(HIDBER, 1999). Last, dimension reduction is another technique for a reduction in the number of random variables. This means that it is the process of reducing data. Some techniques used are Principal Component Analysis (PCA), Latent Semantic Analysis (LSA(LANDAUER; DUMAIS, 1997)]4-, pLSA(HOFMANN, 2013), GLSA), and t-SNE(MAATEN; HINTON, 2008).

Unsupervised learning algorithms have been applied into different networking contexts. For instance traffic classification (LIU; LI; LI, 2007; ERMAN et al., 2007), congestion control (LIU; MATTA; CROVELLA, 2003; BARMAN; MATTA, 2004), and intrusion detection (JIANG et al., 2006; KAYACIK; ZINCIR-HEYWOOD; HEYWOOD, 2003).

2.3 Related Work

Next, we discuss recent studies related to machine learning applied to the networking domain. The studies are divided into two topics: works that leverage machine learning techniques in the control plane and studies that use machine learning techniques in the data plane, and at the end of the second, the contributions of this work are pre-

sented in contrast to the works described. In summary, the Table 2 presents all the works described in this section, the machine learning technique that was applied and the context of application.

2.3.1 Machine Learning in the Control Plane

Recently, with a large amount of data available and the advances in machine learning techniques, the use of these techniques has become increasingly common to solve real-world problems, and consequently has come to the computer networks domain. Tasks like congestion control, load balance, network security, video streaming and others, can now be fulfilled efficiently by machine learning algorithms. This section discusses the works that apply machine learning techniques in the control plane to solve problems in the networks domain showing the advantages of using these techniques.

A popular (but not unique) target for applying machine learning techniques, especially deep learning techniques, is the area of video in general. Video streaming, video analysis, and the various other related areas are among the applications that the most benefit and apply this type of technique.

In the last years, content distribution grew considerably on the Internet. One of the main reasons for that is the growth in the traffic of 360º videos, video-conferencing, live streaming, movies, and videos, even more with all these supporting Ultra-High-Definition (4K), which further increases its volume. It estimates that in 2022 the video traffic represents 82% of global Internet traffic (CISCO, 2020). To ensure video traffic quality, it makes necessary to ensure to use intelligent strategies for communication. For this, recent work is using convolutional neural networks (CNNs) to compress the video in low resolutions without affecting the QoE. Lee et al. (LEE; VENIERIS; LANE, 2020) shows a survey of the state-of-the-art systems that employ neural enhancement. First, the authors present the visual contents delivery systems. They mention systems with adaptative bitrate and neural enhancement and describe systems with different characteristics. For instance, execution of CNN models in client (DASARI et al., 2020) and servers (KIM et al., 2020).

Finally, the authors present the future research directions in neural enhancement to further the benefit of content delivery systems, citing the topics of (I) visual quality, the main challenge in neural enhancement algorithms (II) Efficiency-optimized models, necessary to optimize training and adapt to client computation capacity, (III) image rescaling, ability to work without the high-resolution ground-truth, (IV) meta-learning, optimization is done through an offline pre-train to help in the challenges I and II.

Jaehong Kim et al.(KIM et al., 2020) presents liveNAS, a live video ingest framework for enhancement live video. The liveNAS utilizes super-resolution deep neural networks that employ online training the enhance the live video quality independent of the ingest-side bandwidth. The framework consists of two ingest components. First, the

media-server is responsible for online training and inference. In online training, it learns new features of live video and updates the mapping from low-quality to high-quality. Second, the client ingest is responsible for transmits the encoding video and also some small frames of high-quality captured that are used for the online training of Deep Neural Network (DNN). liveNAS shows the benefits of utilizes super-resolution deep neural networks with online training the enhance the live video quality, reaching an average of 1.16dB overall video quality over WebRTC(state-of-the-art live ingest system), and QoE improvement of 12%-69%.

Real-time video analytics is a complex task, and only is possible been recent advances in computer vision, normally with neural networks-based techniques. However, ANN techniques need a high computation cost, and typically are executed in cloud servers but doing so is challenging due to the high computing and network resource demands of video streaming. So, Yuanqi Li et al. (LI et al., 2020) propose Reducto, a video analytic system that supports real-time on-camera frame filtering. The frames are filtering dynamically, according to the time-varying correlation between feature type, filtering threshold, query accuracy, and video content. Your results show that Reducto can achieve significant filtering (51–97% of frames).

On the other hand, Kuntai Du et al. (DU et al., 2020) advocates that video streaming protocols should be DNN-driven. The authors argue that the inference accuracy depends on the computation featuring in the server-side DNN and cannot be done source (camera). Also, DNN models can provide rich information to guide video streaming. They propose then DDS (DNN-Driven Streaming). The approach follows an interactive workflow, where the source (camera) sends the video segments in low quality to the server, then the server runs DNN and returns feedback with the minimal set of relevant regions necessary for high inference accuracy. Results show that DDS reduces bandwidth usage by up to 59% and improves accuracy by up to 9% with no additional bandwidth overhead.

For network performance, the following works present solutions that apply machine learning techniques for congestion control, traffic analyzes, packet classification, heavy hitters detection and others. First, Solano et al. (ESTRADA-SOLANO; CAICEDO; FONSECA, 2019) propose NELLY: a method for elephant flows detection in Software-Defined Data Center Networks (SDDCN). Your solution executes on the control side and uses an incremental learning approach to provide detection accuracy, low traffic overhead, and the ability to adapt constantly to traffic variations. The results show that your approach is accurate and has a low classification time when uses adaptative decision trees.

Incident routing is a fundamental part of the maintenance of diverse services (e.g., cloud services) and can increase by 10x the time-to-diagnosis due to miss-routings. To solve this problem, Jiaqi Gao et al. (GAO et al., 2020) propose a scouts-based solution. Your approach use per-team scouts, where each teams' Scout acts as its gate-keeper,

being responsible for route relevant incidents to the team, and unrelated ones. Your solution shows are efficient even with the utilization of only a single scout, reducing the time-to-mitigation of 65% of miss-routed incidents.

The packet classification is fundamental for the operation of packets-switched networks. A set of rules performs that classification to determine which activities they should take for each incoming packet. Algorithms for the packet classification implemented on software, focus on two principals strategies: decision-trees and hash-tables. However, those strategies require considerable memory, which makes it difficult your store in the cache. Alon Rashelbach et al. (RASHELBACH; ROTTENSTREICH; SILBERSTEIN, 2020) present the first approach for packet classification that uses the Range-Query RMI machine learning model for accelerating packet classification. The technique called NuevoMatch designs a novel model RQ-RMI which can match keys to ranges, with an efficient training algorithm that does not require exhaustive key enumeration to learn the ranges. The model enables multi-field matching with overlapping ranges through division in an independent set of rules with non-overlapping ranges, called iSets. Your results show that NuevoMatch reduces the memory footprint on average by 4.9X, 8X, and 82x compared to recent researches (CutSplit, NeuroCuts, and TupleMerge, respectively).

The use of ML-based applications for traffic analyzes is already a reality (MIRSKY et al., 2018), (NASR; BAHRAMALI; HOUMANSADR, 2018). In these applications, there are two typical components: a factor extractor and a behavior detector. A factor extractor is responsible for extracts necessary traffic features in network traffic, while a behavior detector utilizes the features extracted for realizing the traffic classification through machine learning algorithms. Note that a behavior detector depends on the factor extractor. However, with the increase of network traffic in the last years (from multi-10s of Gbps to multi-100s of Gbps), there is a growing performance gap for existing traffic analysis applications. Then, Jiasong Bai et al. (BAI et al., 2020) propose FastFE, a high-speed feature extractor that enables the generation of desired traffic features flexibly and efficiently. FastFE extracts feature directly from the data plane through an application that runs in programmable switches, providing a high-level interface that can help network operators express which traffic features they desire.

Since the emergence of TCP, several algorithms have been designed to improve its performance (throughput, lower delay, and fairness). These algorithms, usually, are developed for a specific case (i.e., networks having certain requirements). However, if these assumptions do not hold, the algorithms would not work as expected. In response to this, Abbasloo et al.(ABBASLOO; YEN; CHAO, 2020) propose Orca, an approach to combines classic congestion control strategies and advanced modern deep reinforcement learning (DRL) techniques to introduce a novel hybrid congestion control. Orca utilizes an adaptive control in two levels: fine-grain control and coarse-grain control that enables continuous probing and convergence, besides more efficient and faster training.

In order to improve the performance of machine learning techniques, the following works address the topics of optimization of the training phase, debug and deploy DL systems, build machine learning models, and even the use of machine learning models as a service. Zili Meng et al. (MENG et al., 2020) propose Metis, a framework to interpret diverse deep learning-based network systems. Metis provides a human-readable system for that network operators can easily debug, deploy, and ad-hoc adjust DL-based network systems. Their solution separates the network into two categories: local and global systems. For local systems, Metis applies the decision tree conversion method, while for global systems they formulate with a hyper-graph. Results show that Metis can interpret DL-based systems with high-quality, reaching a QoE improvement of 5.1% on average.

Deep neural networks models have been used to solve networking problems. However, these models have great complexity in their training phase, mainly due to their size – which can take a large amount of time to be completed. Currently, to optimize their training phase, parallel training techniques are generally employed, distributing the training between multiple computing nodes/workers. To carry out this type of training, it has been used stochastic algorithms with gradient compress techniques. However, to perform the communication of quantized gradients without generating network overhead, efficient encoding techniques are needed. To solve this issue Gajjala et al. (GAJJALA et al., 2020) propose three techniques based on the classic Huffman coding for encoding the quantized gradients: run-length Huffman (RLH) encoding, sample Huffman (SH) encoding, and sample Huffman with sparsity (SHS). These techniques seek to explore different characteristics of the quantized gradients during the training phase. Besides, they evaluate their techniques using five different DNN models: ResNet-20, VGG-16, ResNet-50, GoogLeNet, LSTM applied to different scenarios and datasets. To validate their strategies, the authors compare the results obtained with Elias-based code (technique normally used for encoding) and obtained a reduction of data volume in 5.1x(RLH), 4.32x(SH), and 3.8X(SHS).

Building machine learning models is a complex task that usually takes a time-consuming process because running extensive experiments is essential to find better configurations (such as different learning rates or convolution filter sizes). It is even harder when it comes to Deep Learning (DL) models, due to these model's ever-growing architecture size and complexity. In recent years, Apache Spark has become the standard for parallel data processing, where iterative processes are implemented within the bulk-synchronous parallel (BSP) execution model. Then, the BSP model has been used to parallelize building machine learning. However, it is impossible to run asynchronous execution because the BSP has synchronization barriers that prevent this. To solve this problem, Meister et al.(MEISTER et al., 2020) introduce Maggy, an extension to Sparks for run asynchronous machine learning trials. Their solution besides supporting to run

parallel machine learning experiments makes efficient use of parallel computing resources and supports global optimizations. As a result, Maggy reduces the required time of experiments with a fixed number of trials between 33% and 57% when compared with a BSP Spark implementation.

Kourtellis et al. (KOURTELLIS; KATEVAS; PERINO, 2020) argue that machine learning models can be used as a service, but not only in the traditional way, where the applications of machine learning perform centrally with needing to upload all data to the cloud service provider (e.g., Google Cloud (GOOGLE, 2020) or Amazon Web Services (AMAZON, 2020)), but they also be able to work in a distributed way. This method, named Federated Learning (FL) (KONEčNý et al., 2016) is a natural evolution of centralized methods, and enables local training carried out on the user devices. Based on that, the authors propose Federated Learning as a Service (FLaaS), a system that enables collaborative model building in different scenarios and addresses the challenges: privacy and permission management, usability, and hierarchical model training. This system can be executed in different operational environments and supports the building of collaborative models across 3rd-party applications in its FL environment. To validate their system, the authors performed tests using a cell phone configuration (Android) and evaluated the impact of the training on CPU cost, memory footprint, and power consumed, showing the feasibility of their system.

2.3.2 Machine Learning in the Data Plane

Machine-Learning techniques have been widely adopted in the network domain and proved to be efficient in solving a wide range of problems (BOUTABA et al., 2018). However, the execution of these techniques in the control plane causes the resources needed for machine learning algorithms (for example, flow statistics and packet header fields) to be forwarded from the switch to a remote server or host that runs the machine learning algorithms (MCDANEL; TEERAPITTAYANON; KUNG, 2017). This forwarding includes a delay in the packet processing and a large number of flow rules, needing more resources, and making it improbable to process packets at a high-speed.

Since the SDN emergence, also emerged new alternatives to offload the processing of management algorithms of the control plane to the data plane. Thereby is possible to offload partially or even totally the execution of these algorithms. Among those who partially offload, exist works like Hamdan et al.(HAMDAN et al., 2020) that propose the detection of elephant flows (heavy hitters) through a pair of classifiers that share the responsibility of detection. Their technique applies a hybrid classification, where one classifier runs in the data plane and the other runs on the control plane. The authors advocate that the most of mice flows can be detected in the data plane by the SDN switches, decreasing the needed communication with the controller and without affecting the detection accuracy. Their results show that the proposed technique can achieve up to

98.13% accuracy and have a better runtime when compared with other related works.

Then, some works perform the processing of algorithms totally in the data plane, including machine learning techniques. These works show that it is possible to solve several problems through the total execution of the algorithms in the data plane itself. For example Sapio et al. (SAPIO et al., 2017) that proposed DIET, a prototype of a system able to perform offload computation to the data plane. Your initial prototype, implemented in P4, supports a MapReduce application and provides a data reduction of 86.9%–89.3% and a similar decrease in computation time.

Xiong and Zilberman (XIONG; ZILBERMAN, 2019a) introduced IIsy, a software- and hardware-based prototype for in-network packet classification. They explored packet classification through in-network supervised and unsupervised machine learning algorithms (e.g., decision trees, K-means, SVM, and Naïve Bayes) implemented in P4. However, they no investigate the mapping of artificial neural networks in-network, and no consider the training phase of algorithms.

Several efforts have shown the benefits of data plane programmability. This type of programmability enables the in-network execution of diverse specific functions (e.g load balance and packet classification). However, many of these applications require machine learning inference, but the programmable network interface cards often do not support complex operations required by these algorithms, especially operations required by deep neural networks. Therefore, Siracusano et al. (SIRACUSANO et al., 2020) present N3IC, a solution to run neural networks in the data plane with commodity programmable NICs. They show that modern programmable NICs can run NNs, and the execution overhead can be cheaper than the overhead of PCI Express (PCIe) data transfer to the host. Their solution is implemented for two different hardware targets: a System-on-Chip (SoC) based NIC from Netronome, and an FPGA-accelerated NIC and depends on a quantization technique known as binarization. The binarization makes the operations more simple and the computation can be performed using much lighter mathematical operations, offering low memory requirements and causing only a small impact on the inference accuracy. Their implementation can be performed in three modes: leveraging existing NIC programming languages primitives (MicroC for the Netronome NIC and P4 for the NetFPGA), and realizing ANN inference with a dedicated hardware circuitry. Results show that N3IC can perform traffic analysis for millions of network flows per second while forwarding traffic at 40Gb/s.

Siracusano et al. (SIRACUSANO et al., 2018) explore the use of network interface cards for the issue of offloading fully-connected layers processing of artificial neural networks. For this, they implemented toNIC, a system to process binarized fully-connected layers using a commodity SmartNIC. Their results show that through current network cards, it is already possible to process fully-connected layers of binary neural networks.

Siracusano and Bifulco (SIRACUSANO; BIFULCO, 2018) present N2NET, a sys-

tem to run neural networks on a switching chip. Their system shows that current switching chips can run simple neural network models, just like binary neural networks. The authors also suggest that with little additions it will be possible to run models more complex.

Sanvito et al. (SANVITO; BIFULCO, 2018) investigate the possibility of using the programmable network devices themselves as accelerators to run neural network models. The authors advocate that neural network models, principally models with fully connected layers, can improve your efficiency if offloading partially or even totally your processing in the data plane. Then, they develop an initial prototype that executes ANN models quantized on network processor-based SmartNICs and programmable switching chips.

Analyzing storage cost and latency required to training BNN models, Guan et al. (GUAN et al., 2019) propose a new deep learning model, called Recursive Binary Neural Network (RBNN), which aims to decrease data storage required in the training phase keeping a high classification accuracy and low latency. Your solution implemented on the FPGA platform uses incremental training and data storage recycling, reducing the off-chip data access. The results show that your model can reduce 4-6x data storage requirements keeping the classification accuracy as compared with conventional BNN training.

li et al. (LI et al., 2018) proposed INCEPTIONN (In-Network Computing to Exchange and Process Training Information Of Neural Networks), a solution for accelerating distribute training of deep neural networks. The proposed solution combines hardware and algorithmic innovations. The results show it can reduce the communication time between neurons by 70.980.7% and offers 2.23.1X speedups over the conventional training system, with the same level of precision.

For DNN models that use reinforcement learning for your training, Li et al. (LI et al., 2019) present a solution to accelerate the distributed reinforcement learning. Your solution called iSwitch performs the acceleration in-switch (data plane) moving the gradient aggregation from the server nodes to the network switches. The solution reduces the end-to-end communication overhead and supports synchronous and asynchronous distributed RL training, with a speedup of 3.66x and 3.71x, respectively.

Qin et al.(QIN et al., 2020) propose the use of BNNs for intrusion detection at the network edge through adopting a federated learning approach for BNN training. Their goals are to provide a scalable intrusion detection strategy that gives high accuracy with low memory and communications costs and an architecture based on programmable switches to run on edge devices able to packet classification while updating the learning models. Furthermore, the authors developed a prototype in P4 for testing your strategy in a real scenario. The results show that their method can line-speed packet processing while achieving a high classification accuracy, low false alarm rate, and small communications overheads.

Although machine learning techniques are efficient to solve several problems in various areas of the network, if executed in the control plane, they still have some limita-

tions, such as the need to communicate with an external entity (e.g., server or host) and the overhead generated by the communication. Nevertheless, many works still use these techniques in the control plane, as can be seen in Table 2.

Offloading the execution of these machine learning techniques to the data plane proved to be possible and can solve these limitations, but it is still little explored. Works like (XIONG; ZILBERMAN, 2019b) only encompass the execution of simple machine learning models, and without considering their training phase, while on the other hand jobs like (GUAN et al., 2019) and (LI et al., 2019)only address how to optimize the training phase of more complex models. Fortunately, N2net (SIRACUSANO; BIFULCO, 2018) and BaNaNa (SANVITO; BIFULCO, 2018) take the first step towards the execution of complex machine learning models (ANNs) in-network. ANNs can be performed in the data plane by adapting to simpler models, such as BNNs (SIRACUSANO et al., 2018). However, these works are still limited to the execution of only one neural network in the entire network, making infeasible the scenario where we can have several neural networks running simultaneously in the network, each solving a different task. Also, these works carry out the executions of the neural networks in a single network device, which can lead to a degradation of the performance of this device, due to the complexity of the execution of these models.

In this work, by contrast to the works presented in Table 2, we encompass the mapping of multiple ANNs in-network. Other than that, we carry out the mapping of each neural network in a distributed way, with only one neuron running per network device, but allowing the communication between neurons (synapses) and the correct operation of the neural network. Mapping multiple ANNs in-network allows that multiple problems to be solved simultaneously, and the distributed mapping of neural networks allows for a distribution of the required workload, not penalizing just a part of the network. Both the cited topics not are covered in recent works.

Table 2 – Overview of related literature.

Work	**ML technique**	**Control/Data Plane**	**Application/Context**
(KIM et al., 2020)	DNN	Control Plane	Live video
(MENG et al., 2020)	Decision Tree and hypergraph	Control Plane	Framework to interpret DL systems
(GAO et al., 2020)	Random Forest CPD+	Control Plane	Cloud analyses and diagnosys
(DU et al., 2020)	DNN	Control Plane	Video Streaming
(GAJJALA et al., 2020)	DNN	Control Plane	Optimize training phase
(MEISTER et al., 2020)	DNN	Control Plane	Optimize build ML models
(KOURTELLIS; KATEVAS; PERINO, 2020)	Federated Learning	Control Plane	Ofert ML as a service
(LI et al., 2020)	DNN	Control Plane	Video analytics
(LEE; VENIERIS; LANE, 2020)	NN	Control Plane	Survey of NE in content delivery systems
(ABBASLOO; YEN; CHAO, 2020)	Deep reinf learning	Control Plane	Framework to congestion control
(RASHELBACH; ROTTENSTREICH; SILBERSTEIN, 2020)	NN	Control Plane	Packet Classification
(ESTRADA-SOLANO; CAICEDO; FONSECA, 2019)	Decision Trees	Control Plane	Elephant flows detection
(BAI et al., 2020)	-	Control Plane	Feature extractor for traffic analyzes
(HAMDAN et al., 2020)	-	Control and Data	Heavy hitters detection
(SIRACUSANO; BIFULCO, 2018)	BNNs	Data Plane	Implementation in current switch chips
(SIRACUSANO et al., 2018)	BNNs	Data Plane	Implementation Fully-connecteds layers of BNN in SmartNiCs
(XIONG; ZILBERMAN, 2019a)	Decision Trees, K-Means, SVM and Naive Bayes	Data Plane	Packet Classification
(LI et al., 2018)	DNNs	Data Plane	Acelerating Deep Neural Networks Trainning
(QIN et al., 2020)	BNNs	Data Plane	Intrusion detection
(SAPIO et al., 2017)	-	Data Plane	Map Reducee Aplications
(SIRACUSANO et al., 2020)	BNNs	Data Plane	Run Neural Networks in data plane
(SANVITO; BIFULCO, 2018)	BNNs	Data Plane	Run BNNs on SmartNICs
(GUAN et al., 2019)	RBNN	Data Plane	Optimize BNN tranining
(LI et al., 2019)	DNN	Data Plane	Accelerate the reinforcement learning for DNN models

3 In-Network Neural Network

In this chapter, we present the In-Network Neural Networks problem. We start by providing the problem overview, followed by its formal definition. Then we introduce a constructive heuristic and a math-heuristic approach to solve the problem efficiently.

3.1 Problem Overview

For the sake of clarity, we describe the In-Network Neural Network Problem (IN^3-P) without any mathematical rigor. Given a set of ANNs (neurons and interconnections) and physical infrastructure (nodes and links), the IN^3-P asks for a valid mapping between each ANN and the underlying network infrastructure.

In short, a feasible solution should map each neuron of each ANN to a single programmable device without any sharing (we further explain one exception). The IN^3-P holds this assumption because data plane virtualization (Saquetti et al., 2020) – which would allow neurons to effectively share forwarding devices – is still in its infancy, and we can not yet operationalize such settings seamlessly. However, note that as soon as data plane virtualization becomes a reality, this restriction can be softened. Given that we have a feasible mapping to all neurons of ANNs, we have to ensure that all inputs/outputs of neurons are propagated to their interconnected neighbor neurons. To avoid extra transit of packets on the network, we embed the neurons' input/output information on the available space of network flow packets (i.e., using the same principle of in-band network telemetry – INT). Given a pair of mapped neurons, we ensure that there are available network flows between mapped neurons (with available capacity) so as neurons' input/output can hitchhike on.

Figure 8 illustrates a feasible solution for the IN^3-P problem. Observe that an ANN might assume any arbitrary topology (i.e., from simple perceptron to deep convolutional networks). For the example, we consider a three-layer ANN with seven neurons (N_1–N_7) and twelve interconnections. On the right, there is a feasible mapping of each neuron to a single programmable device. Due to space constraints, we highlight only three neurons' interconnections. First, note that the interconnection between neuron N_1 and N_4 is mapped to the forwarding devices A and B, respectively. Observe that the output of neuron N_1 is embedded on a packet (1) that goes from A to E. When the packet arrives on a forwarding device B, the data plane obtains the information from neuron N_1 and processes its output to the next set of neurons. As soon as neuron N_4 receives the information from neurons N_1 and N_2, it can process and send to the output layer (2), which is composed by neurons N_6 and N_7. It is important to mention that many optimization goals could be considered and that we opt to minimize the distance between neurons of adjacent layers – i.e., between input/hidden layers and between hidden/output layers. The reason is to minimize synchronization issues and timeouts on the implementation.

The time it takes for each neuron to synchronize its input data can cause an

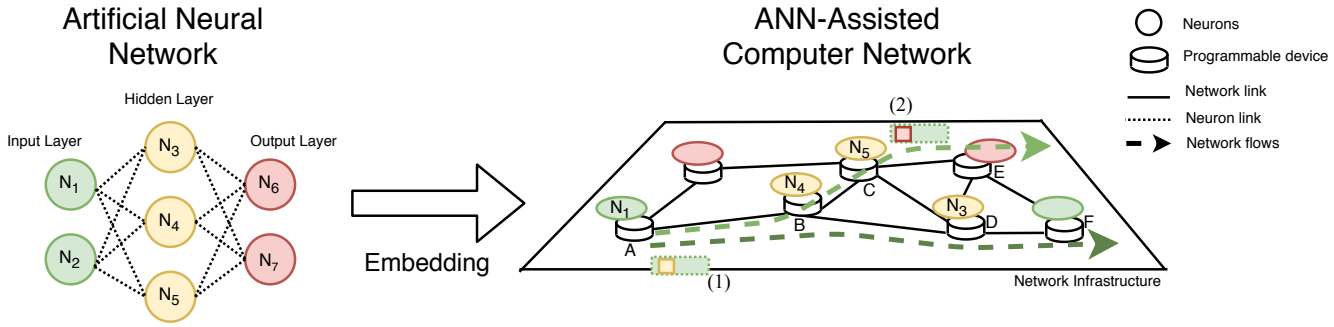


Figure 8 – Example of a artificial neural network being provisioned into a programmable network infrastructure.

impact on ANN execution, degrading your performance and even making it impossible to function.

The synchronization time is the time that a neuron has to wait between receiving the first and the last input data. For example, a neuron that receives input data from three other neurons in another layer needs that the three data to be received to start executing. So the synchronization time is the difference between the receiving time of the first and last input data. This time difference is related to the distance that the adjacent neurons are positioned in the network. This is because the data is propagated using the network flows (i.e., telemetry concept), then the distance between the network devices (and other aspects of the path, like available bandwidth and delay) is responsible for the communication time between neurons.

Figure 9 shows an ANN with four neurons, one in the input layer (N1), two in the intermediate layer (N2 and N3), and one in the output layer (N4), and two possible mappings of this ANN in the same network infrastructure. Note that in this example, we do not consider the constraint of active network flows to interconnect neurons so, only links between devices are needed. Assuming that all links have the same cost (in this case, assuming that each hop takes a time unit), in the mapping (A) we need 1 time unit for the N1-N3 interconnection and 3 time units for the N1-N2 interconnection. If the processing of each neuron also takes a 1 time unit, in the time unit 4 the neuron N4 will already have the result of N3 (1 time unit to N1-N3 + 1 time unit to run N3 + 2 time-units to N3-N4), but will have to wait 3 more time units to get the result of N2 and start executing (At that moment N2 just only finished your execution, because 3 time units to N1-N2 + 1 to run N2). Finally, after 7 time units, N4 can start executing, and the processing of the ANN with mapping (A) is finished after 8 time units. By contrast to mapping (A), in the mapping (B) after 1 time unit, all neurons in the hidden layer can already begin to perform (1 time unit to N1-N2 and 1 time unit to N1-N3, which occur simultaneously). Then, at the end of time unit 3, the neuron N4 can start executing, ending the processing of the ANN in 4 time units.

Through this example, we can see how the orchestration of the ANNs mapping

can influence the performance of its execution, where the execution with the mapping (A) takes twice the time compared to the mapping (B). In addition to the increase in execution time, large distances and asynchrony between the positioned neurons may cause data to become unusable (the data takes so long to arrive that it is no longer useful, as it no longer represents the current state of the network) and timeout (where a device receives new data from a neuron while still waiting for an earlier response from another neuron).

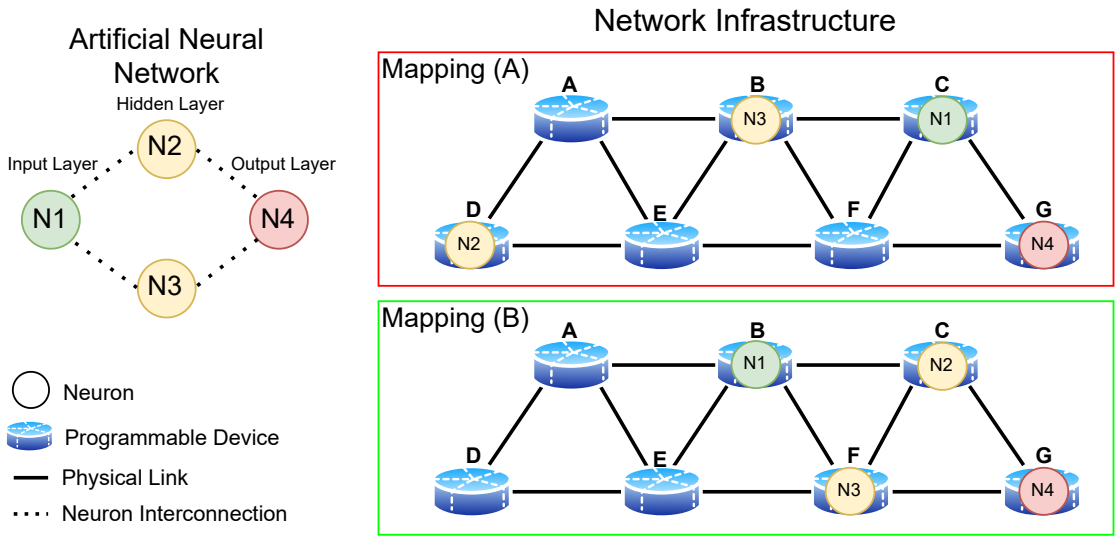


Figure 9 – Example of two valid mappings to a simple ANN

Despite being similar to the well-studied Virtual Network Embedding (VNE) (FISCHER et al., 2013), the IN^3-P has different restrictions and goals that make the problem even more intractable. The IN^3-P considers that ANN interconnections move neurons' outputs through active network flows. One might think that a reduction from IN^3-P to VNE instances would solve the problem at hand. For example, consider a network infrastructure where links only exist *if* there exists an active network flow between a given pair of forwarding devices. In this way, we would ensure that whenever two neurons are mapped, there would have a network flow to move output data to/from forwarding devices. Although this might represent a relaxed solution (lower bound), we miss the ability to pick-up and deliver neuron output to forwarding devices on the network flow route. For instance, in Figure 8, network flow from A to E moves neuron N1 output until forwarding device B, then collects neuron N4 output and moves to forwarding device E. Thus, we would substantially reduce the search space and lead VNE models/algorithms to come up with a sub-optimal or infeasible solution.

Therefore, in brief, in this work we model and develop algorithms for the IN ^{3} - P problem, which aims to map in the network infrastructure a set of ANNs in a distributed way (see Figure 10). To perform this mapping, active network flows are used to satisfy the interconnections between neurons and layers, taking advantage of techniques such

as in-band telemetry. In addition, it seeks to minimize the maximum distance between interconnected neurons positioned in the network, to optimize their performance and minimize synchronization issues. For example, in Figure 10, with a ANN and a network topology with a set of active flows, a valid mapping for the ANN is obtained, obeying all the restrictions described above, which will be formally described in the next section.

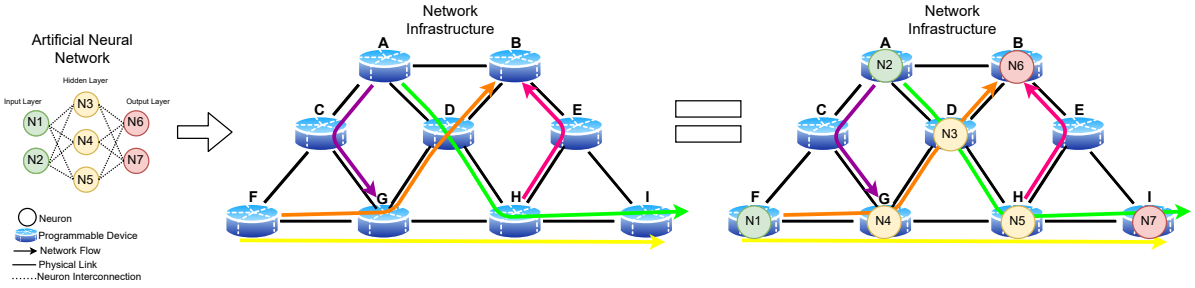


Figure 10 – Example of a valid mapping

3.2 Model Description

The proposed optimization model considers a physical network infrastructure $G = (D, L)$, a set of n specialized artificial neural networks $A = (A_1, A_2, ..., A_n)$ and a set of data plane telemetry states V. Set D in network G represents programmable forwarding devices $D = \{1, ..., |D|\}$, while set L consists of unidirectional links interconnecting pair of devices $(i, j) \in (D \times D)$. Similarly to the recent literature (HOHEMBERGER et al., 2019), we consider that each programmable forwarding devices $d \in D$ is able to embed a subset of in-band states $V_d \subseteq V$ into networks packets (which are used by the input layer). Each telemetry state $v \in V$ has its size defined by function $S : V \rightarrow N^+$.

Artificial neural network $A_n \in A$ is represented by a k-partite graph $A_n = (N_n^k, I_n)$, that is, a graph whose vertices are partitioned into k different independent sets. Note that each independent set N_n^k represents neurons of the k-th layer. Similarly, set I_n represents unidirectional interconnections among neurons, i.e. $(i, j) \in (N_n \times N_n)$. As previously mentioned, ANN inputs are network telemetry statistics collected from the data plane (e.g., queue occupancy, processing time, or any other information). Neurons from the input and output layers play an important role in how ANNs work. Input neurons are in charge of feeding the ANN with appropriate network information, while output layers are the result that might be tied to a particular forwarding device. For instance, an input neuron might collect the data plane processing time of an access router. Therefore, it can not be mapped to an arbitrary forwarding device. Location requirement of neurons (e.g., input layer) is defined as $L : N_n^k \rightarrow D$. We assume each neuron from the input layer (i.e., $k = 1 : N_n^k$) has only one input requirement (e.g., data plane processing time). In case more network statistics are required from the same network device (e.g., data plane processing and queue occupancy), we model it as different neurons. Set R define network

statistics requirement as a tuple $R : N \rightarrow (D \times V)$. For example, $R(n_1) = i, v : (i \in D, v \in V_i)$. For simplicity, we index each tuple element by $R^d(n_1)$ and $R^v(n_1)$ to refer to the network device and telemetry item, respectively. It is worth mentioning that the model allows representing any ANN topology.

We assume that there exists a set of active network flows F on top of network infrastructure G. Packets of network flows F are used to collect real-time data plane telemetry statistics (from input layer neurons) and interconnect neurons from different layers. A flow $f \in F$ has two endpoints (*i.e.*, ingress and egress forwarding devices) and is routed through the network infrastructure G using a simple path $\mathcal{P}$. We denote the path taken by flow f as function $\mathcal{P} : F \rightarrow \{D_1 \times ... \times D_{|D|}\}$. We consider that network flows $f \in F$ are encapsulated in a forwarding protocol (*e.g.*, NSH[1], IPv4) and, therefore, the amount of available space to embed data plane telemetry items $v \in V$ (or neurons' output) in packets is bounded by a given constant $\mathcal{C}_f \in N^+$.

Given the problem input, the IN^{3}P optimization model seeks a feasible assignment of neurons to programmable devices, while satisfying ANN inter-layer communication. Next, we describe the integer linear programming formulation for the problem. Given a network infrastructure G, a set of ANN A, a set of network flows F, and a set of telemetry items V, the optimization problem seeks a feasible solution that minimizes the distance between mapped adjacent ANN layers. The objective functions is described in Equation (1), considering $\mathcal{M}(i, j)$ the distance between all pair of forwarding devices $(i, j) \in (D \times D)$. The model output is denoted by a 3-tuple $\chi = \{X, Z, Y\}$. Variables from $X = \{ x_{i,j,n,s,t}, \forall (i, j) \in L, n \in N, (s, t) \in I_n\}$ indicates that interconnection (s, t) from ANN n is using physical link (i, j) from G. Variables from $Z = \{ z_{s,t,n,f}, \forall (s, t) \in I_n, n \in N, f \in F\}$ indicates that interconnection (s, t) from ANN n utilizes network flow f to encapsulate its data. Last, variable $Y = \{ y_{i,n,s}, \forall i \in D, n \in ANN, s \in N_n\}$ indicates that a neuron s from ANN n is mapped to forwarding device i. Next, we describe the MILP formulation for the IN^{3}P problem.

$$\textbf{Minimize} \quad \sum_{n=1}^{N} \sum_{(s,t) \in I_n} \sum_{(i,j) \in L} w_{i,j,n,s,t} \cdot \mathcal{M}(i, j) \tag{3.1}$$

Subject to:

$$\sum_{f \in F} \sum_{\substack{j \in D: \\ (i,j) \in \mathcal{P}(f)}} x_{i,j,n,f,s,t} - \sum_{f \in F} \sum_{\substack{j \in D: \\ (j,i) \in \mathcal{P}(f)}} x_{j,i,n,f,s,t} = y_{i,n,s} - y_{i,n,t} \quad \forall i \in D, n \in N, (s, t) \in I_n \tag{2}$$

$$x_{i,j,n,f,s,t} \leq z_{s,t,n,f} \quad \forall (i, j) \in L, n \in N, (s, t) \in I_n, f \in F \tag{3}$$

[1] <https://tools.ietf.org/html/rfc8300>

$$\sum_{f \in F} z_{s,t,n,f} \leq 1 \qquad \forall n \in N, (s,t) \in I_n \quad (4)$$

$$\sum_{i \in D} y_{i,n,s} = 1 \qquad \forall n \in N, s \in N_n \mid R^d(s) = \emptyset \quad (5)$$

$$\sum_{i \in R^d(s)} y_{i,n,s} = 1 \qquad \forall n \in N, s \in N_n \mid R^d(s) \neq \emptyset \quad (6)$$

$$\sum_{n \in N} \sum_{s \in N_n | R^d(s) \neq \emptyset} y_{i,n,s} \leq 1 \qquad \forall i \in D \quad (7)$$

$$\sum_{n \in N} \sum_{(s,t) \in I_n} x_{i,j,n,s,t} \cdot \mathcal{S}(R^v(s)) \leq \mathcal{C}_f \qquad \forall f \in F, (i,j) \in \mathcal{P}(f) \quad (8)$$

$$y_{i,n,s} + y_{j,n,t} \leq \frac{w_{i,j,n,s,t}}{2} \qquad \forall n \in N, (s,t) \in I_n, (i,j) \in D \times D \quad (9)$$

$$x_{p,i,j} \in \{0,1\} \qquad \forall p \in P, (i,j) \in L \quad (10)$$

$$z_{p,v,i} \in \{0,1\} \qquad \forall p \in P, v \in V_i, i \in D \quad (11)$$

$$y_{i,n,s} \in \{0,1\} \qquad \forall i \in D, n \in N, s \in N_n \quad (12)$$

$$w_{i,j,n,s,t} \geq 0 \qquad \forall (i,j) \in L, n \in N, (s,t) \in I_n \quad (13)$$

Constraint set (2) refers to the traditional flow conservation constraint. This constraint set ensures that a well-formed path between mapped neurons. Note a valid path between mapped neurons has at least one network flow $f \in F$ that is routed through physical link (i,j). Constraint sets (3) and (4) ensure that an ANN interconnection utilizes the same network flow to move data from one neuron to another. Constraint set (3) keeps track of used network flows by ANN interconnections. In turn, constraint set (4) limits the number of used network flows to be one. It is worth mentioning that this constraint might be softened and allow multiple networks flows to transport inter-layer communications. Despite the benefit of enlarging the search space, it hinders operationalizing in-network ANNs in real-environments. Constraint sets (5) and (6) ensures that all neurons are mapped to a programmable data plane. Note, whenever a neuron has a location requirement (i.e., $R^d(s)$), constraint set (6) ensures that each neuron to be mapped to a single forwarding device. On the contrary, equation set (5) ensures neurons are being mapped to a valid programmable device. Constraint set (7) ensures that neurons do not share programmable devices. This constraint is valid to all neurons that do not have location constraints – otherwise, constraint set (7) is not applicable. In turn, constraint set (8) ensures that network flows $f \in F$ have enough space to carry ANN inter-layer communication. Constraint set (9) correlates the mapped positions of the neurons to an auxiliary variable w, which controls the distance between the neurons. Last, constraint sets (10)–(13) define the domains of output variables.

Algorithm 1 Overview of the constructive heuristic.

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: network infrastructure, $\mathcal{R} = (\mathcal{L}, \mathcal{N}, \mathcal{I})$: neural networks, $\mathcal{F}$: network flows, T_{total}: attempts limit
Output: χ: best solution found to the constructive method

$\chi \leftarrow$ initial empty solution
while T_{total} is not exceeded **do**
  $\chi_{current} \leftarrow$ new empty solution
  for each neural network $r \in \mathcal{R}$ not positioned **do**
    $l \leftarrow$ layer most connected from r
    $V_{aux} \leftarrow$ all routers from $\mathcal{V}$ without neurons mapped
    $\chi'_{current} \leftarrow$ maps layer l on the most connected routers from V_{aux}
    for each layer $l \in \mathcal{L}$ not positioned **do**
      for each neuron $(n \in \mathcal{N}) \in l$ **do**
        $\mathcal{D} \leftarrow$ dependencies with all neurons positioned in $\chi'_{current}$
        $\mathcal{P} \leftarrow$ list of possible routers to position neurons according to $\mathcal{D}$
        if $\mathcal{P} \neq \emptyset$ **then**
          $p \leftarrow$ random router from $\mathcal{P}$
          $\chi'_{current} \leftarrow$ maps neuron n in the router p
        else
          $\chi'_{current} \leftarrow$ infeasible solution
        end if
      end for
    end for
  end for
  if $\chi'_{current}$ better χ **then**
    $\chi \leftarrow \chi'_{current}$
  end if
end while
return χ

3.3 Proposed Approaches

3.3.1 Constructive Heuristic

Algorithm 1 shows the process of creating a constructive solution used as the basis for Algorithm 2. The algorithm's objective is to generate an initial valid solution, whether it is complete (with all ANNs mapped) or not. In this case, a feasible solution means it meets all the constraints of model 3.2, except constraints 5 and 6, i.e., the algorithm does not need to map all ANNs. However, the ANNs mapped must be according to restrictions. Besides, the algorithm seeks to minimize the objective function through a heuristic approach, based on positioning the neurons of the most connected layers of the ANNs in the devices that have more flows. Because these layers has the neurons that most interconnect with other neurons. So, maximizing the flows available to these neurons makes it easier for them to perform these interconnections over a short distance.

As input, the algorithm receives a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ representing the network infrastructure, a set of neural networks $\mathcal{R} = (\mathcal{L}, \mathcal{N}, \mathcal{I})$ where $\mathcal{L}$ represents the layers, $\mathcal{N}$ the neurons and $\mathcal{I}$ the interconnections, a conjunct $\mathcal{F}$ of network flows, and some attempts

Algorithm 2 Overview of the math-heuristic

Input: T_{local}: time limit for each solver run, $\mathcal{M}$: optimization model
Output: χ: best solution found to the optimization model
1: $\chi \leftarrow$ initial solution generated by constructive heuristic
2: $\mathcal{M} \leftarrow$ set model constraints with solution χ
3: $\mathcal{C} \leftarrow$ generation all combinations of ANNs mapped in χ
4: **for** each combination $c \in \mathcal{C}$ **do**
5: $\quad \mathcal{M}' \leftarrow$ remove combination c from model $\mathcal{M}$
6: $\quad \chi' \leftarrow$ solution of model $\mathcal{M}'$ by the solver under time T_{local}
7: $\quad$ **if** χ' better χ **then**
8: $\quad\quad \chi \leftarrow \chi'$
9: $\quad\quad \mathcal{M} \leftarrow$ set model constraints with new solution χ
10: $\quad\quad \mathcal{C} \leftarrow$ generation all new combinations of ANNs mapped in new solution χ
11: $\quad$ **end if**
12: **end for**
13: **return** χ

limit T_{total}, and as an output, the algorithm returns the best solution found during the execution of all attempts T_{total}.Initially, line 1 creates an empty-solution. Line 2 causes the algorithm to execute up to the number of attempts defined. In line 3, a new empty-solution is created for the current attempt. In line 4, the algorithm runs through the entire R set of ANNs for mapping. Then, in lines 5 and 6, we select the layer most connected to the current neural network and the list of available neurons for first mapping. Then, the selected layer is positioned on the available routers that participate in the highest number of active network flows (line 7). For each remaining layer and neuron (line 8-9), find all dependencies (interconnections between current neuron and positioned neurons) and from there find possible routers to perform positioning (routers that have active network flows that pass through routers where neurons involved in dependencies are positioned)(line 10-11). If there are neurons in P to perform the mapping, choose one randomly and follow the algorithm. Otherwise, continue with a partial solution (without mapping the ANN) (lines 12-15). Finally, it tests whether the solution obtained in that attempt is better than the current best global solution, and exchanges updates to the best global solution if necessary(lines 21-22). Therefore, the complexity of the algorithm is given by $\mathcal{O}(\mathrm{T}_{total} \cdot \mathcal{R} \cdot \mathcal{N} \cdot \mathcal{L})$, where $T_{total}$is the attempts limiter, $\mathcal{R}$ the neural networks, $\mathcal{L}$ the layers of neurons, and $\mathcal{N}$ the neurons of each neural network.

3.3.2 Math-heuristic Approach

Algorithm 2 presents an overview of the proposed strategy. The strategy used is based on the creation of an initial solution χ (based on Algorithm 1)(line 1), for then, use of a subset of ANNs $\mathcal{R} = (\mathcal{N}, \mathcal{L}, \mathcal{I})$ mapped in this solution χ to set some $Y_{s,n,i}$ variables of the model 3.2, while the other variables remain unchanged. Assigning variables simplifies the model, avoiding the need to find a solution for all variables, but only for

those not assigned. This assignment of variables to the model is only possible because, as mentioned in Section 3.3.1, the initial solution χ generated by Algorithm 1 has ANNs mapped according to model 3.2.

After assigning all $Y_{s,n,i}$ variables in the model (line 2) $\mathcal{M}$, the algorithm generates the combinations of neural networks to be removed from the model at each interaction(lines 3, 4). The generation of the combinations of neural networks removed from the $Y_{s,n,i}$ variables is through the generation of simple combinations between the set of all ANNs mapped in solution χ. As a result, only combinations of $1 \leq size \leq$ {Number of mapped ANN -1 } , are used, to always leave at least one ANN assigned to the model. In addition, combinations are sorted increasingly by size so that combinations removing fewer networks run first.

Then, the algorithm removes the ANNs referring to the current combination ⌋ of model $\mathcal{M}$(line 5). Next, the algorithm uses the IBM Cplex solver to solve $\mathcal{M}$(line 6). Then, after the model is solved, the solution χ' generated from the current combination of assigned $Y_{s,n,i}$ variables is compared to the current solution χ. If the generated solution χ' is equal to or worse than the current solution χ, the model $\mathcal{M}$ is restored with the current solution and left for the following combination. If the generated solution $\chi'$is better than the current solution χ(line 7), the current solution χ is updated to the generated solution χ'(line 8), and a new set of combinations $\mathcal{C}$ from the new current solution(lines 9, 10) is generated. The algorithm terminates when there are no more combinations in $\mathcal{C}$ to be attributed to $Y_{s,n,i}$, returning the best solution found.

3.3.3 Random Algorithm

The random algorithm (see Algorithm 3) aims to generate a valid initial solution with the help of random choices. The algorithm receives as input a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ representing the network infrastructure, a set $\mathcal{R} = (\mathcal{L}, \mathcal{N}, \mathcal{I})$ of neural networks, a set $\mathcal{F}$ of network flows, and two attempts limiters T_{total} and T_{perANN}. The limiter T_{total} limits the number of attempts that will be made to find the solution, and the limiter T_{perANN} defines the maximum number of times that we will try to map an ANN within an existing solution. The T_{perANN} limiter is necessary because the algorithm has random choices, then invalid solutions are frequently generated, requiring some attempts to find a valid mapping. On the other hand, T_{total} is necessary because even if it manages to map some networks, sometimes complete mappings are not found for all available ANNs, requiring some attempts to find a complete mapping with many ANNs mapped as possible.

The idea of the algorithm is to select a neural network randomly and then a neuron from that network also at random and find the possible routers that this neuron can be mapped (that can be done in constant time using hash tables for flows and interconnections). Once all possible routers are found, the algorithm randomly selects one of these routers and maps the neuron. The process is repeated for all neural networks until a valid

Algorithm 3 Overview random algorithm

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: network infrastructure, $\mathcal{R} = (\mathcal{L}, \mathcal{N}, \mathcal{I})$: neural networks, $\mathcal{F}$: network flows, T_{total}: attempts limit, T_{perANN}: attempts limit per neural network
Output: χ: best solution found to the random method

```
1: χ ← initial empty solution
2: while T_total is not exceeded do
3:     χ_current ← new empty solution
4:     for all r ∈ R do
5:         r ← a random neural network ∈ R
6:         while T_perANN is not exceeded do
7:             while r is not mapped or is possible to map do
8:                 n ← a random neuron ∈ r
9:                 V ← a list of possible routers to map n
10:                if V ≠ ∅ then
11:                    v ← a random router ∈ V
12:                    mappingNeuron(χ'_current, n, v)
13:                end if
14:            end while
15:        end while
16:    end for
17:    if χ'_current better χ then
18:        χ ← χ'_current
19:    end if
20: end while
21: return χ
```

mapping is found or the counter reaches T_{perANN}. Each time a complete solution is found, it is compared with the best current solution, to save the best mapping found. Then the algorithm is terminated when the number of trials T_{total} runs out. Then, the complexity of the algorithm is given by $\mathcal{O}(\mathrm{T}_{total} \cdot \mathrm{T}_{perANN} \cdot \mathcal{R} \cdot \mathcal{N})$, where T_{total} and T_{perANN} are the attempts limiters, $\mathcal{R}$ the neural networks, and $\mathcal{N}$ the neurons of each neural network.

3.3.4 Greedy Algorithm

The greedy algorithm (see Algorithm 4) is very similar to the random, receiving the same inputs, but with an important difference in its decisions. Rather than randomly selecting a router from between the possible ones, the algorithm weighs the routers with the maximum distance it will include in the solution and then selects the router with the shortest distance (with a random tie). This decision in the choice of routers to perform the mapping aims to generate solutions with a shorter maximum distance between neurons, seeking to optimize the mappings and minimize synchronization problems, as discussed in Section 3.1. In addition, weighting routers by the distance included in the solution does not change the complexity of the algorithm, as it can be done with the simple addition of a field in the hash table of flows. Therefore, similar to the random algorithm, the complexity of the algorithm is given by $\mathcal{O}(\mathrm{T}_{total} \cdot \mathrm{T}_{perANN} \cdot \mathcal{R} \cdot \mathcal{N})$, where T_{total} and T_{perANN} are the

Algorithm 4 Overview greedy algorithm

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: network infrastructure, $\mathcal{R} = (\mathcal{L}, \mathcal{N}, \mathcal{I})$: neural networks, $\mathcal{F}$: network flows, T_{total}: attempts limit, T_{perANN}: attempts limit per neural network
Output: χ: best solution found to the random method

```
1: χ ← initial empty solution
2: while T_total is not exceeded do
3:     χ_current ← new empty solution
4:     for all r ∈ R do
5:         r ← a random neural network ∈ R
6:         while T_perANN is not exceeded do
7:             while r is not mapped or is possible to map do
8:                 n ← a random neuron ∈ r
9:                 V ← a list of possible routers to map n
10:                if V ≠ ∅ then
11:                    V ← weigh the routers with the max distance included
12:                    v ← router with minimal distance ∈ V
13:                    mappingNeuron(χ'_current, n, v)
14:                end if
15:            end while
16:        end while
17:    end for
18:    if χ'_current better χ then
19:        χ ← χ'_current
20:    end if
21: end while
22: return χ
```

attempts limiters, $\mathcal{R}$ the neural networks, and $\mathcal{N}$ the neurons of each neural network.

4 Evaluation

Thus chapter presents the evaluation of the results obtained with the strategies proposed in this work. Section 4.1 describes the environment, parameters and metrics used in the evaluation, Section 4.2 presents the baseline used in the experiments, and the Section 4.3 presents and discusses the results obtained.

4.1 Workload

We run the proposed model using International Business Machines (IBM) *CPLEX Optimization Studio* 12.9 to obtain optimum solutions and implement the proposed heuristic approach in Java. The experiments were performed on a machine with Advanced Micro Devices (AMD) Threadripper 2920X processor and 80 GB of Random Access Memory (RAM), using the Ubuntu 16.04 operating system. Different instances of network infrastructures were generated with fat-tree topology (PETRINI; VANNESCHI, 1997), varying its size in 20 ($k = 4$) and 80 ($k = 8$) network devices. The fat-tree topology is a typical data center topology, and represents an ideal environment for executing in-network ANNs.

The fat-tree topology is organized according to k core routers. From that k, it is calculated the number of core, aggregation, and edge switches, with k being a power of 2. The number of core switches is $= (k/2)^2$, and they are interconnected with k pods which contain $k/2$ aggregation switches and $k/2$ edge switches each. Each aggregation switch is connected to $k/2$ core switches and $k/2$ edge switches. Edge switches connect to $k/2$ aggregation switches and $k/2$ servers. For the easy of presentation, Figure 11 illustrates an example of a Fat-Tree topology using a $k = 4$.

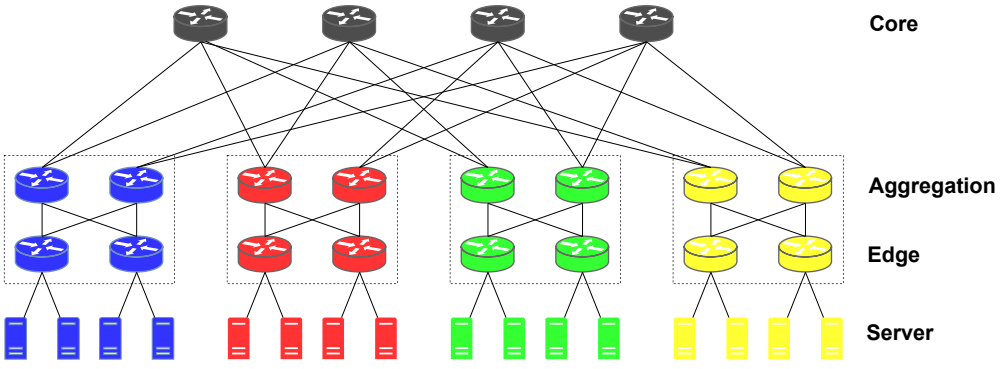


Figure 11 – Example of fat tree with $k = 4$

The active network flows were generated randomly in range 10 to 100 (10 out of 10) always interconnecting two edge switches, and with capacity generated randomly in range 0 to 10. The number of available ANNs (to be mapped) varies from 2 to 16, and each ANN has five neurons divided into 3 layers, with 1 in the input layer, 3 in the middle

layer, and 1 in the output layer. Further, we vary the ability to share data plane by neurons (number of neurons that can be mapped on a switch) from 1 to 4. That data plane sharing means that constraint set (7) is being relaxed – that is, it is not limiting anymore the mapping of only one neuron per network device. Finally, the parameters T_{total} and T_{perANN} of Algorithms 3 and 4 are varied from 200 to 1000 (200 in 200).

We focus the evaluation on four main metrics: (i) the maximum distance (longest distance between any two interconnected neurons positioned on the network), (ii) the number of ANNs positioned on the network, (iii) the number of network flows utilized by our approach, and (iv) the time taken to find a solution.

4.2 Baseline

As a baseline comparison for the designed algorithms, we used an adaptation of the VNE (FISCHER et al., 2013) problem model. The VNE is a problem similar to the problem we are solving in this work. However, in the VNE model, there are no network flows, only vertices and edges. Therefore, we can reduce an IN^3-P instance to a VNE one.

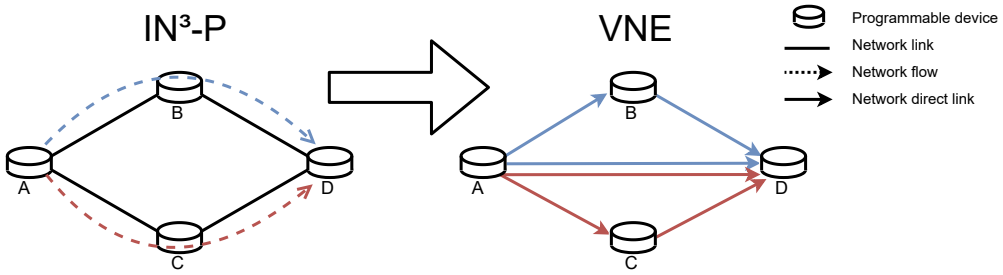
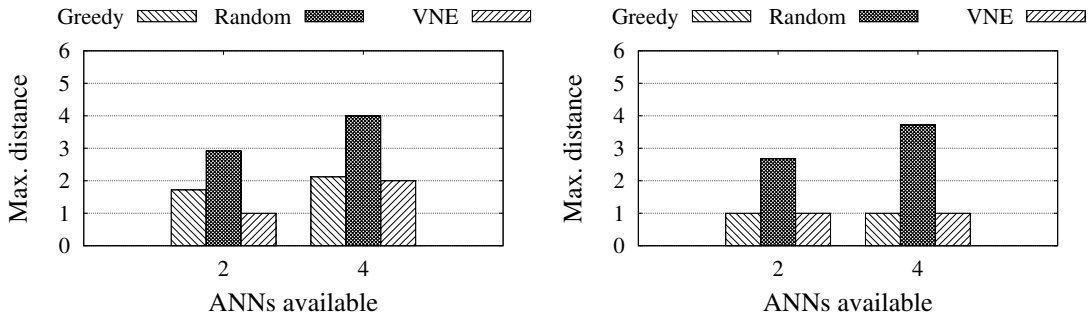


Figure 12 – Reduction from an IN^3-P instance to a VNE instance

Figure 12 shows the process of reducing an IN^3-P instance to a VNE instance. To achieve this reduction, we must transform an IN^3-P graph (with vertices, edges, and network flows) into a VNE graph (with only vertices and edges), but both represent the same instance of the problem. To perform the reduction, we firstly copy all vertices that participate in some network flow in IN^3-P. After that, we interconnect with directed links all pairs of vertices that are part of the same network flow, maintaining the flow directions. For example, we have a network flow that passes through devices A, B and D (in that order), so now we have edges directed from A to B, A to D, and B to D.

4.3 Results

In this section, we present and discuss the results obtained in the experiments. Firstly, in Section 4.3.1 we assess the solution's quality obtained by the proposed approaches. For that, we consider as a criteria the maximum distance and number of ANNs



(a) Sharing = 1 (b) Sharing = 4

Figure 13 – Max. distance according to ANNs available to k = 4

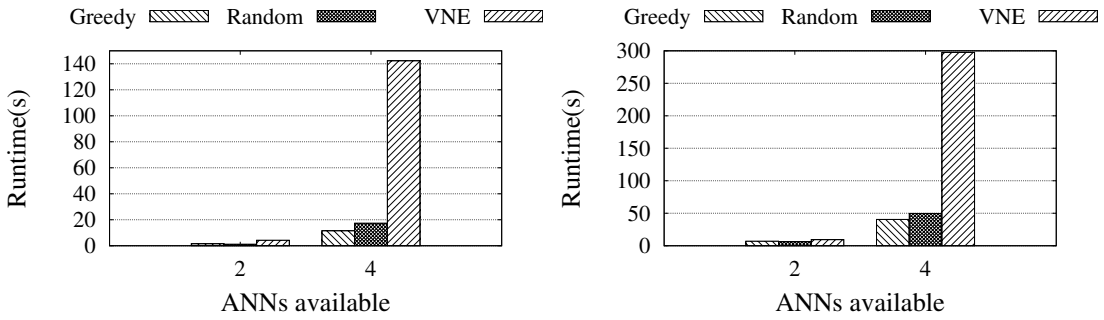
mapped. Secondly, in Section 4.3.2, we assess the impact of using an increasing number of network flows by our solutions. Thirdly (Section 4.3.3), we evaluate the flow utilization by the solutions and its incurred impacts. Then, in Section 4.3.4 we verify the impact of parameters T_{total} and T_{perANN} on the solution's quality. Finally, in Section 4.3.5 we discuss the time spent by each algorithm to generate a feasible solution according to the number of ANNs.

4.3.1 Quality of Solutions

In this subsection, we evaluate the quality of the solutions generated by analysing the following metrics: (i) max. distance and (ii) number of ANNs mappeds. The first evaluation (Figure 13) illustrates the maximum distance between neurons in a topology with 20 devices (k = 4), varying the number of ANNs from 2 to 4 (maximum possible with sharing = 1) and with two levels of neuron sharing: sharing = 1 (Figure 13a) and sharing = 4 (Figure 13b). For the sake of clarity, whenever a parameter is not mentioned, it means that it has been varied in all cases described in Section 4.1, and the result shows the average of the obtained results.

We start this evaluation with Sharing = 1. This is the hardest case for mapping ANNs, since we can only map one neuron per device, requiring more devices and network flows (discussed after, in Section 4.3.3) for coming up with a feasible solutions. On the other hand, when considering Sharing = 4 , it represents the more relaxed scenario (and, probably easier to solve), as it allows four neurons to be mapped on top of the same device. In both cases, we can observe that the VNE model was able to map all available ANNs with a maximum distance (in average) less than the greedy and random strategies. This is mainly because the VNE model (computed using IBM Cplex solver) returns the best possible solution for the instance that has been adapted to. The greedy and random algorithms are simple and focus on scalability, seeking for a feasible solution in a lower runtime.

The VNE achieves better results in Figure 13, however, the number of resources needed for this (see Figure 14) makes it non-scalable even for small cases. The figure shows



(a) Average (b) Maximum

Figure 14 – Runtime for k=4 and sharing = 1

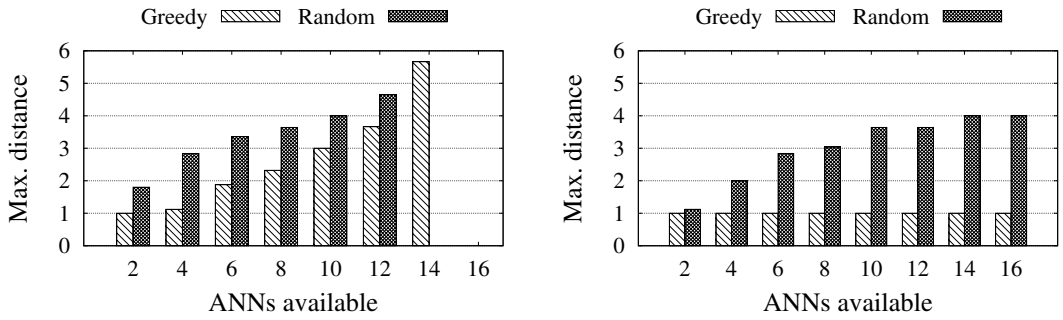
the average (Figure 14a) and maximum (Figure14b) time used by the strategies to solve a single instance of the problem. We can observe that on average the VNE takes 12.3x more time than the greedy, 8.15x more time than the random, and more than 6x times longer for both strategies in the worst case. In addition, the memory requirements for the VNE is larger than the available amount in our setup in order to run larger instances (e.g. $k = 80$). The same issue also occurs in the experiments of the math-heuristic, which uses the model 3.2 implemented in the IBM Cplex. However, the model 3.2 is far more complex that the VNE model (see Section 3.1), and it is not possible to finalize all its execution – even for the topology with k = 4. Therefore, from this point, we continue the evaluation with only algorithms 3 and 4, that can be executed in our available environment.

Then, considering the algorithms 3 and 4, we can see that the greedy strategy can map all available ANNs with a maximum distance (in average) lower than the random strategy. This is because the greedy strategy makes mapping decisions based on distance, while the random strategy randomly. In addition, we can observe the difference between the easiest case (sharing = 1, Figure 13a) and the hardest case (sharing = 4, 13b) of mapping, with both algorithms having greater maximum distances in Figure 13a.

Next, in Figure 15 we performed the same evaluation for infrastructure with 80 devices (k = 8), varying the number of ANNs from 2 to 16 (maximum possible with sharing = 1 and k = 8). Also, we performed the same evaluation considering hardest mapping case (1 sharing, Figure 15a) and the easiest case (4 sharing, Figure 15b).

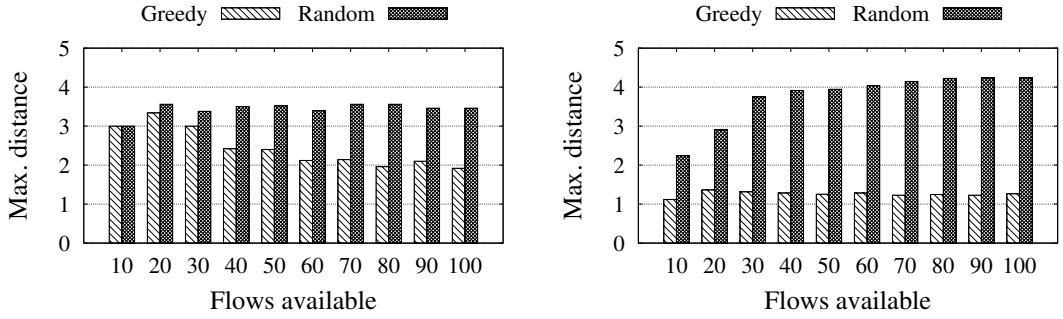
We can observe again that in both cases the greedy algorithm can map the ANNs with a maximum distance shorter than the random algorithm. In addition, note that in Figure 15a none of the algorithms can find a valid mapping for 16 ANNs, and for 14 ANNs only the greedy algorithm can do the mapping. Furthermore, in Figure 15b (as well as in Figure 13b) we can see that the greedy algorithm manages to take better advantage of the sharing of devices, maintaining a constant mapping distance for all the number of ANNs while the Random algorithm increases its mapping distance.

The fact that the greedy algorithm obtains mappings in cases where the random does not do so is because it always tries to map neurons on closer devices. So it benefits



(a) Sharing = 1 (b) Sharing = 4

Figure 15 – MaxDist according to ANNs available to k = 8



(a) Sharing = 1 (b) Sharing = 4

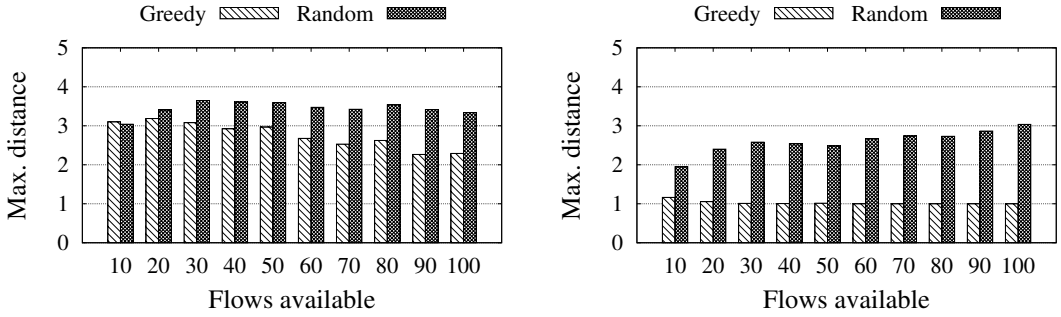
Figure 16 – Max. distance according to the increase in the network flows to k = 4.

from the fact that there might be more chances to exist network flows between neighbor devices than between distant devices. Consequently, these choices also shorten the maximum distance – as depicted by our experiments so far.

4.3.2 Flows' Impact

In this subsection, the objective is to evaluate the impact that the increase in the number of network flows causes on the solution. First, we evaluate the impact of increasing network flows over the maximum distance. Then, Figure 16 shows the maximum distance obtained for the flow variation from 10 to 100 for the topology with k = 4, separating again the hardest case (sharing = 1, Figure 16a) and the easiest case (sharing = 4, Figure 16b).

So we can see that for Sharing = 1 (Figure 16a), the greedy algorithm benefits from the increase in the number of flows, while random maintains a similar distance. For Sharing = 4 (Figure 16b), the greedy algorithm achieves mappings with low distance from 10 flows, and maintains a similarity with the increase in the flows, while the random increases the maximum distance according to the increase in the flows. This is because the random does not make decisions based on distance, which means that increasing the number of flows increases the possibility of interconnection with more distant devices, which can now be selected by random to map neurons. This does not affect the greedy



(a) Sharing = 1 (b) Sharing = 4

Figure 17 – Max. distance according to the increase in the network flows to k = 8.

algorithm as it will prioritize mapping on neighbor devices.

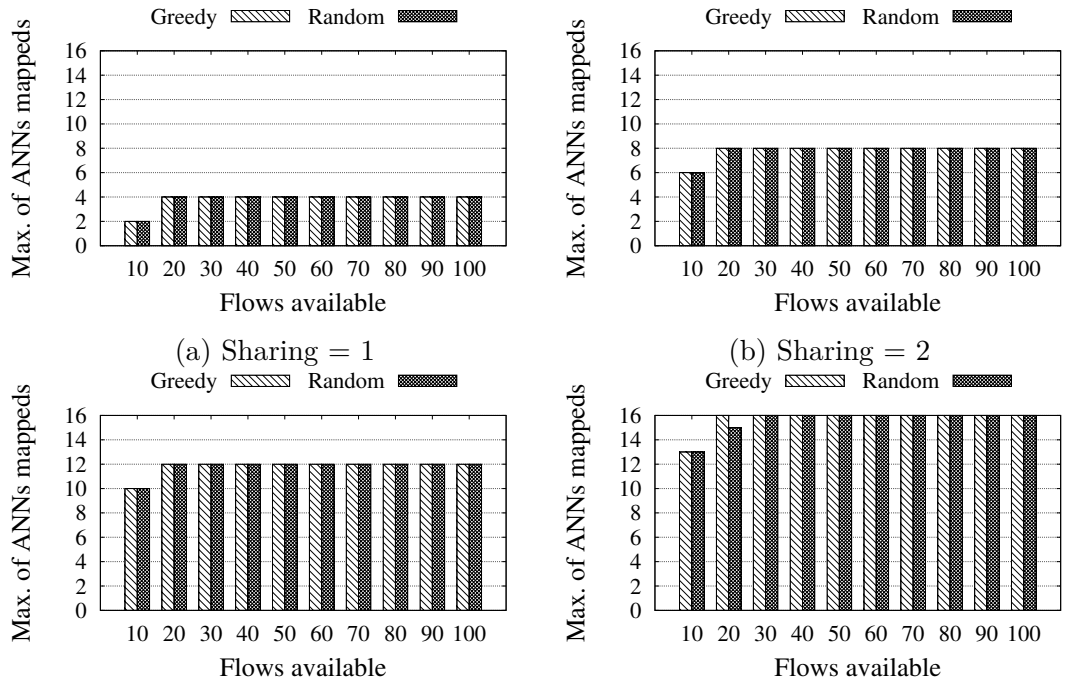
Similarly, Figure 19 presents the same evaluation but considering k = 8. Also, for Sharing = 1 (Figure 17a), the greedy algorithm can benefit from increased flows while the random has similar behavior. For Sharing = 4 (Figure 17b), the greedy algorithm finds a mapping with a low distance from 10 flows and maintains it. On the other hand, the random algorithm has small increases in the distance according to the increase in the flows.

Note that for both network sizes the behaviors are similar. In addition, we can see that the random algorithm has similar behavior for Sharing = 1, and increasing behavior for Sharing = 2. This happens because by increasing the sharing of devices from 1 to 4, we increase the possibilities of mapping in 4x, increasing the solution space that the algorithm will move away from the shortest distances since it does not make decisions based on them. In addition, the greedy algorithm gradually decreases the distance according to the increase in the number of flows, except in cases where it reaches a distance close to 1 (best possible distance).

Then, in Figures 18 and 19 we evaluate the maximum number of ANNs that can be mapped according to the increase in the number of flows. In Figure 18, we can see the maximum number of mappings obtained for the topology with k = 4, varying the share from 1 to 4. Note that the number of possible ANNs to be mapped increases according to the increase in share (e.g., 4, 8, 12, and 16 for sharing 1, 2, 3, and 4 at k = 4), and can be defined by calculation $ANNsPossibleToMap = SizeNetwork * Sharing/ANNsLength$.

In Figure 18, we can see that for k = 4, in all cases of sharing (Figures 18a, 18b, 18c and 18d) the maximum number of ANNs possible can already be mapped from 20 flows for both algorithms (except for the random in the case 16b, which needs 30 flows). This means that for this topology, no more than 20 network flows are needed when the objective is to map the maximum number of ANNs possible. This is due to the fact that a k = 4 generates a small topology, with only 20 network devices, making the number of flows tested to be significant in relation to the size of the infrastructure.

Then, in Figure 19 we can see the same test for a topology with 80 devices (k =



(a) Sharing = 1

(b) Sharing = 2

(c) Sharing = 3

(d) Sharing = 4

Figure 18 – Number max of ANNs mappeds according to the increase in the network flows to k = 4.

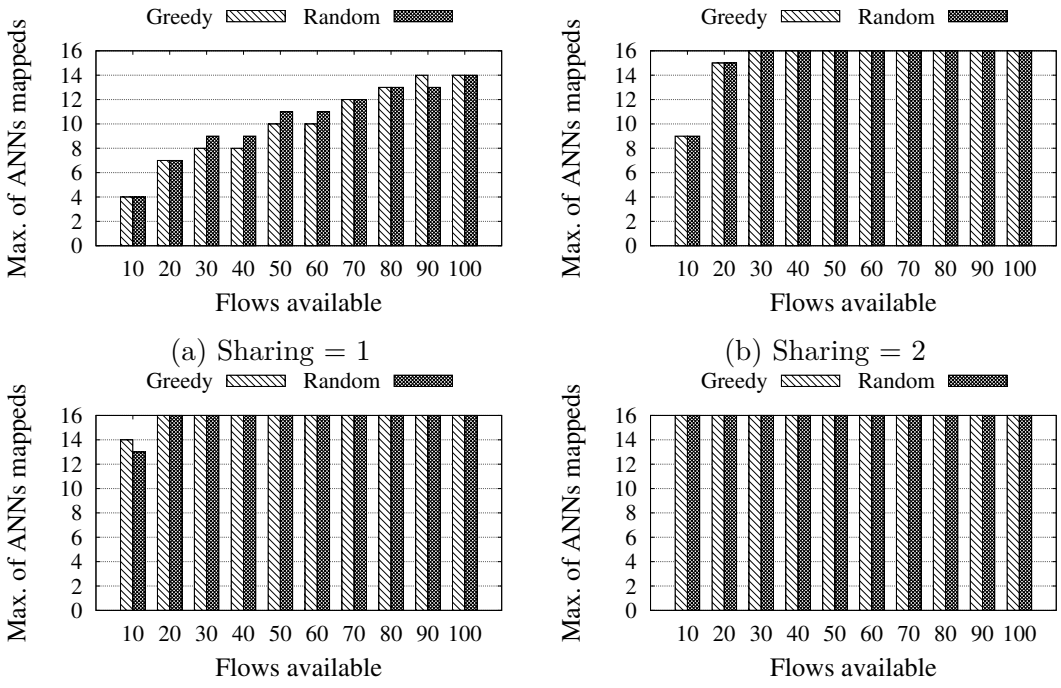
8). In this case, for sharing = 1 (19a) both algorithms benefit from increasing the number of flows to map more ANNs, reaching 14 mappings (2 less than the maximum possible), and intercalating which (algorithm) obtains more mappings per case.

However, for Sharing = 2, it is possible to map the 16 ANNs with 30 flows, and in the other cases (Sharing = 3 and Sharing = 4), Figures 19c and 19d) from 10 flows. The difficulty in finding mappings in this case 19a is because the number of flows tested is no longer as significant about the size of the topology as previously. In addition, increasing the size of the topology and the number of ANNs to be mapped increases the problem complexity.

4.3.3 Flows' Utilization

In this topic, we evaluate the number of flows used to perform the mapping of ANNs. The use of flows is an important metric for measuring how much the mapping of ANNs can interfere with the network traffic.

Figure 20 shows the relationship between the number of ANNs mapped and the average flows used to perform these mappings. The figure shows results for the 4 sharing options (1, 2, 3 and 4) to assess the impact of sharing on the use of flows. So, we can see that for Sharing = 1 (Figure 20a) we even use 52 network flows (average) to map 14 ANNs. In addition, the greedy algorithm used more flows than the random algorithm in all mapped ANNs values. This is because when you cannot share a device with more than



(a) Sharing = 1

(b) Sharing = 2

(c) Sharing = 3

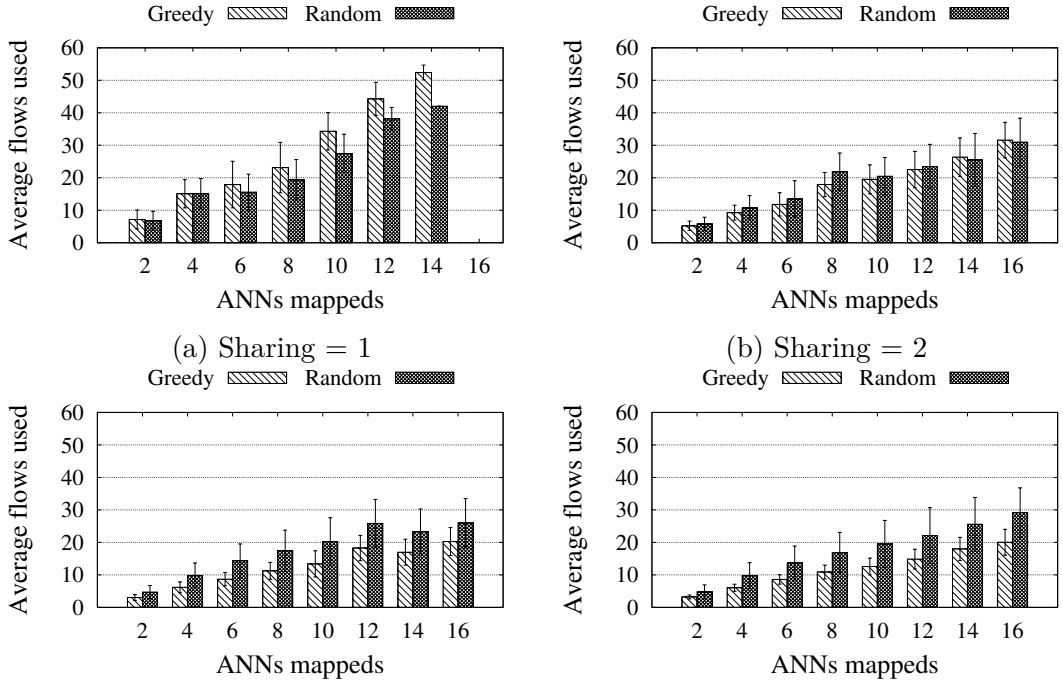
(d) Sharing = 4

Figure 19 – Number max. of ANNs mappeds according to the increase in the network flows to k = 8.

one neuron, the maximum distance is directly conditioned by the use of flows (see Figure 21a), and as the greedy algorithm seeks to minimize the maximum distance, it uses more flows.

Nonetheless, when we can share devices with more than one neuron (Figures 20b, 20c, and 20d) the use of flows tends to decrease according to the increase in sharing (see Figure 21b) because neurons from the same ANN that are positioned on the same device do not need to use network flows to perform communication, and there is no distance to that interconnection. Therefore, with the sharing of devices, the use of flows decreases and the greedy algorithm starts to use fewer flows than the random algorithm (see Figures 20b, 20c and 20d). Because, to decrease the maximum distance, it positions the largest possible number of neurons of the same ANN in each device. Figure 21 show exactly this behavior, because with sharing = 1 (Figure 21a) we decrease the maximum distance through the use of more flows, and with the share = 4 (Figure 21b) we decrease the distance through the use of fewer flows, mapping neurons from the same ANNs on the same devices.

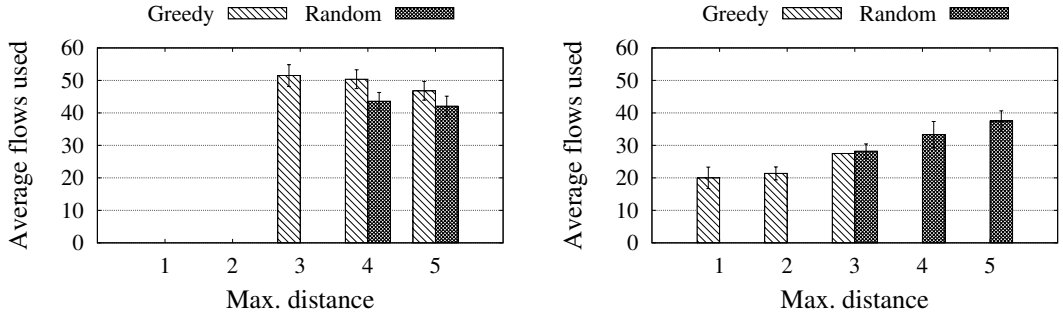
Note that the Figure 21 shows the relationship between flows used and max. distance considering the mapping of the maximum possible ANNs per sharing (for example, for Sharing = 1 it was possible to map 14 ANNs, so we consider 14 as the fixed number of neural networks to be mapped in the test). This justifies the fact that there are no bars of the two algorithms at all points of max. distance because for example the random algorithm is not able to reach a distance = 1 or 2 for 16 ANNs (max with sharing = 4),



(a) Sharing = 1 (b) Sharing = 2

(c) Sharing = 3 (d) Sharing = 4

Figure 20 – Average of flows used per ANNs mapped



(a) Sharing = 1 (b) Sharing = 4

Figure 21 – Average of flows used per max. distance

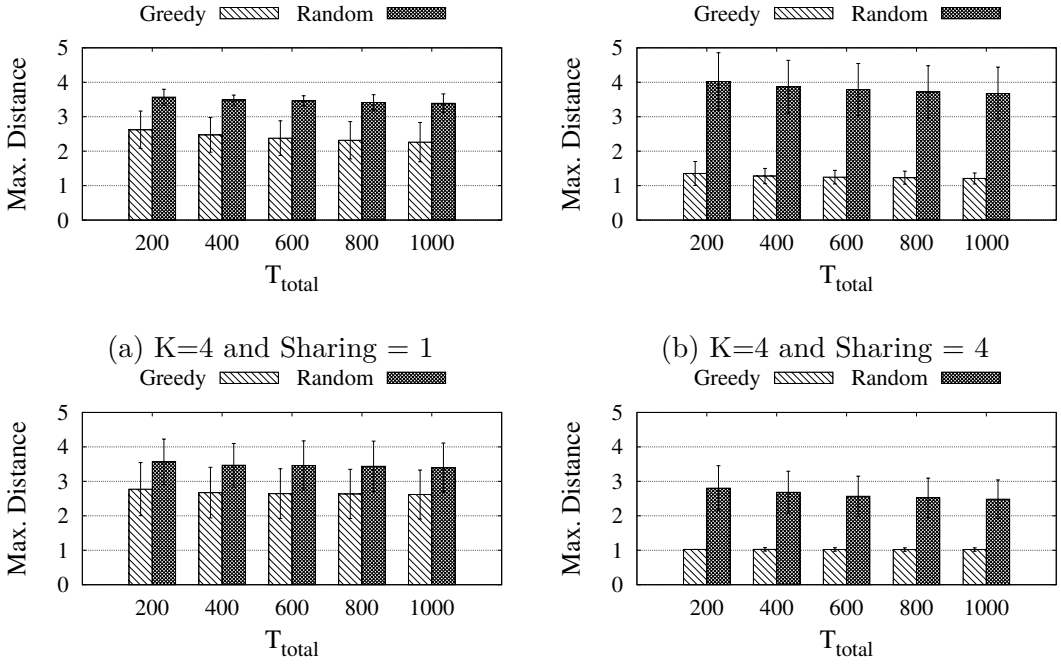
or distance=3 for 14 ANNs (max with sharing = 1).

4.3.4 Parameter Adjustment

In this subsection of the evaluation, the objective is to assess the impact of the T_{total} and T_{perANN} parameters on the generation of the solutions. These parameters are used in Algorithms 3 and 4 (see Sections 3.3.3 and 3.3.4) as attempt counters, to control the attempts employed on obtaining solutions.

First, we evaluate the impact of the T_{total} parameter, which defines the number of attempts used to find a complete solution (a solution that maps all available neural networks). Figure 22 shows the variation of this parameter for the values of k = 4 and 8, and shares = 1 and 4.

Then, from the analysis of the figure we can see that by increasing T_{total} from 200



(a) K=4 and Sharing = 1

(b) K=4 and Sharing = 4

(c) K=8 and Sharing = 1

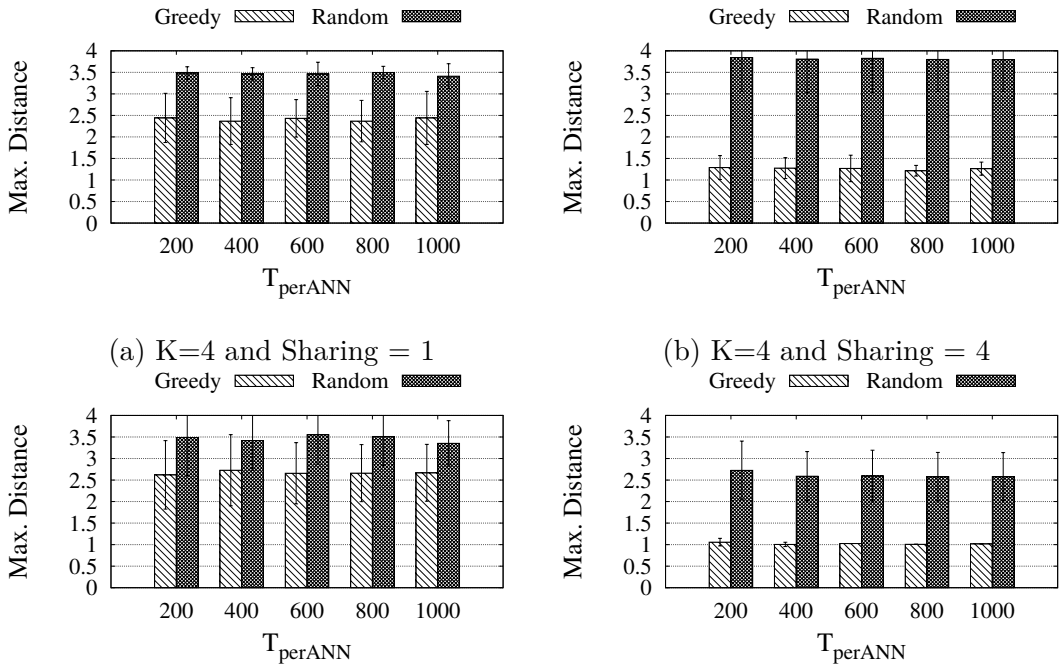
(d) K=8 and Sharing = 4

Figure 22 – Impact of T_{total} parameter

to 1000, we have some gains in decreasing the maximum distance. For $k = 8$ we have $\approx -12.9\%$ for the random algorithm and 4 shares, and $\approx -5\%$ for both algorithms for sharing = 1. In K = 4, with share = 1 we have $\approx -12.5\%$ and $\approx -9\%$ for greedy and random algorithms (respectively), and $\approx -15.9\%$ and $\approx -5.3\%$ for greedy and random algorithms(respectively) to sharing = 4.

After that, we evaluate the increase of the parameter T_{perANN} also from 200 to 1000. This parameter defines the number of times we will try to map (in case of failure) an ANN within a solution. The Figure 22 show the results for $k = 4, 8$ and shares = 1,4. For this parameter, we highlight the $\approx -5.8\%$ of the random algorithm, in the topology with k = 8 and 4 shares. Apart from this result, the others results varied below 5%, with the majority remaining constant, decreasing only the standard deviation.

Therefore, by the evaluations of the two parameters, we can see that the T_{total} parameter influences and improves the results more than the $T_{perANNs}$ parameter, that got little influence on the quality of the solution. Note that the standard deviation of each case is also influenced by the other parameters of the execution (e.g., network flows and T_{total} for $T_{perANNs}$ and vice-versa). To realize this evaluation the average was calculated, varying all the other parameters not fixed in the graph, which causes the standard deviation to increase.



(a) K=4 and Sharing = 1

(b) K=4 and Sharing = 4

(c) K=8 and Sharing = 1

(d) K=8 and Sharing = 4

Figure 23 – Impact of T_{perANN} parameter

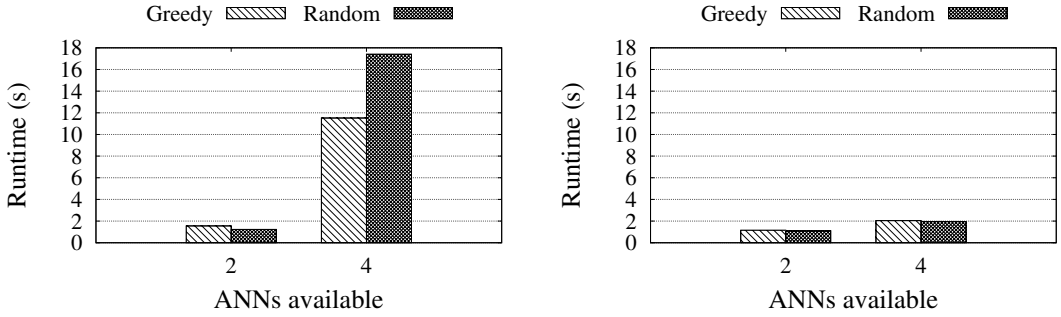
4.3.5 Time Cost

Finally, in the last evaluation section, we discuss the runtime required by the algorithms to perform the ANN mappings. We present results for the two topology sizes (20 and 80 devices) and with device Sharing = 1 and Sharing = 4. In addition, as we evaluated in the previous section (Section 4.3.4) the impact of T_{tota} and T_{perANN} parameters on the quality of the solutions generated, here we evaluate how much they impact the runtime so we can have an idea of the cost-benefit of increasing these parameters.

First, we evaluate the execution time spent in mapping the 4 possible ANNs for the topology of k = 4, with 1 and 4 shares. Through Figure 24 we can see that using sharing = 1 (Figure 24a), in addition to generating solutions with greater distances, makes the creation of a slower solution when compared to the solution with sharing = 4 (Figure 24b), taking up to 8.5x more to map 4 ANNs.

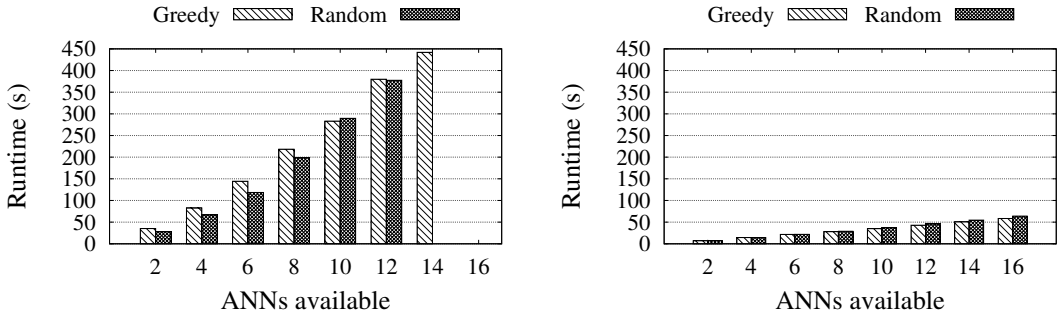
Similar to what occurs in Figure 25, with k = 8, where the case with sharing = 1 (Figure 25a)is approximately 8.65x slower than the case with sharing 4(Figure 25b). This difference occurs due to the complexity of carrying out the mapping when the sharing is equal to 1 (as discussed in Section 4.3.1), with sharing = 4 if there are many mapping possibilities, facilitating the creation of the solution.

After that, we started to evaluate the impact of parameters T_{total} and T_{perANN} on the execution time of the solutions. Then, Figure 26 shows the execution time of parameters T_{total} and T_{perANN} for a topology with k = 4. Analyzing this figure, we can see that although the parameter T influences the quality of the generated solutions more,



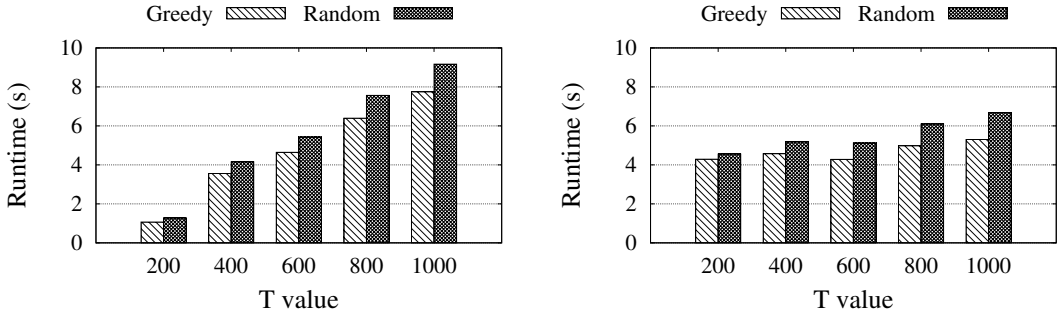
(a) Sharing = 1 (b) Sharing = 4

Figure 24 – Runtime per number of ANNs and k = 4.



(a) Sharing = 1 (b) Sharing = 4

Figure 25 – Time of execution per number of ANNs and k = 8.

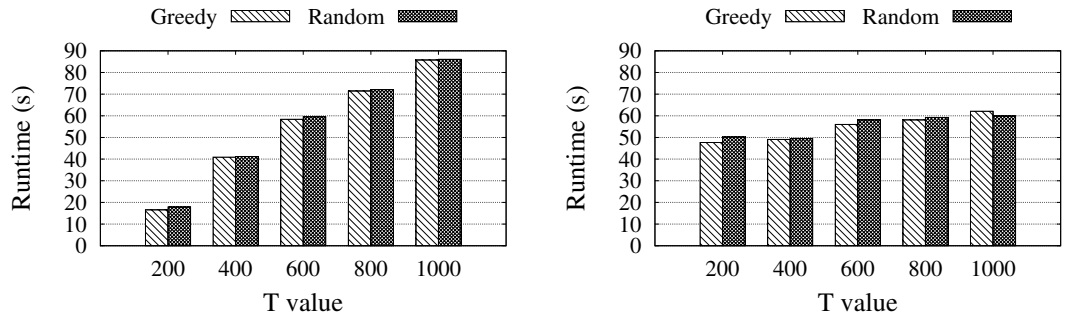


(a) T_{total} (b) T_{perANN}

Figure 26 – Time of execution T parameters and k = 4.

it also influences the execution time, increasing the execution time by almost 10x from 200 to 1000. In contrast to this, the T_{perANN} parameter initially has a higher execution time, however it does not grow as much as T_{total}. This longer time in the first cases of the parameter T_{perANN} is because it considers the average time of all variations of T_{total} (for example, T_{perANN} is fixed at 200, and the result is the average of $T_{perANN} = 200$ and T_{total} varying from 200 to 1000). Therefore, the fact that T_{perANN} does not obtain great variation with the different fixed values, means that it has a low influence on the execution time.

Similarly, for the topology with k = 8 (Figure 25) The behavior of both parameters remains, with T_{total} having a smaller impact on the initial time, but growing a lot with



(a) T_{total} (b) T_{perANN}

Figure 27 – Time of execution T parameters and k = 8

the increase of the parameter. In addition, we have the T_{perANN} which has a larger initial impact, but which does not grow as much as the parameter increases, not reaching a 30% increase, while the T_{total} it has almost 500%.

To sum up, after evaluating the execution time of all these cases, we can observe that: (i) the sharing of neurons impacts both the quality of the solution and the execution time; (ii) despite having improvements in the solution according to the increase in the T_{total} parameter, this increase also has a great impact on the execution time; and (iii) the T_{perANN}, despite not contributing much to the improvement of the solution, ends up contributing (albeit in a lesser way than T_{total}) in the execution time of the solution.

5 Final Remarks

In this chapter, we provide an overview of the main points covered throughout this work. In addition, we review the results and contributions of this work. Then, to finish this work, we describe and present possible future work related to the further steps, issues not addressed in the scope of this work, and solutions to limitations.

In this work, we present an optimization problem to the mapping of Artificial Neural Networks in-network, namely In-Network Neural Network Problem. Albeit the IN3-P is similar to the well-studied Virtual Network Embedding Problem (FISCHER et al., 2013), it has different restrictions and objectives which makes its resolution ineffective with algorithms used in VNE, turning it more difficult to solve. To demonstrate these differences and propose a solution to the problem, we made an adaptation IN3-P that can reduce an instance to the VNE problem and then seek a solution. In addition, we present the following algorithms to solve the problem: (i) a constructive algorithm that seeks to find an initial solution to the problem, (ii) a mathematical heuristic that uses this initial solution to find even better solutions, (iii) an algorithm pseudo-random that seeks to find a feasible solution to the problem, and (iv) a greedy algorithm that seeks to find solutions with a low communication overhead. Our results show that it is unfeasible to solve the problem in a scalable way with the adaptation to the VNE and the proposed mathematical heuristic due to the high amount of resources (memory and processing) required by these methods. However, our other algorithms are capable of mapping in a distributed way (with only one neuron per network device) up to 4 ANNs in a fat-tree network size 20 (maximum number of possible mappings) and up to 14 ANNs in a fat-tree network with a size 80 (only two ANNs less than the maximum possible). In addition, the algorithms can find the maximum mapping for 20 devices (4 ANNs) with only 20 network flows available and a maximum distance (on average) very close to 1 (minimum possible distance for mappings). Then, we can describe as the main contributions of this work: (i) The formalizations of the In-Network Neural Network through an optimal Integer-Linear Programming model, (ii) the development of techniques capable of finding efficient and scalable solutions for the problem, (iii) the evaluations of the impact of the device sharing by neurons, and the impact of the available/used flows in the generated mappings, for works such as (Saquetti et al., 2020) and (SANVITO; BIFULCO, 2018), which share the concept of performing the processing (at least part) of ANNs in-network.

5.1 Achievements

This work has benefited from previously published works, reusing ideas and concepts (with for example in-band telemetry).

1. Castro et al. Análise do Desempenho de Heurísticas na Coleta de Informações de Telemetria In-Band. In: 17^a Escola Regional de Redes de Computadores (ERRC

2019) (CASTRO et al., 2019)

2. Rumeningue et al. Orchestrating in-band data plane telemetry with machine learning. In: IEEE Communications Letters, 2019 (HOHEMBERGER et al., 2019)

3. Castro et al. Patcher: Towards fault-tolerant probing planning for in-band network telemetry. In: IEEE Latin-American Conference on Communications(LATINCOM) (CASTRO et al., 2020)

5.2 Future Work

In this section, we present possible future directions for this work, including topics not covered in the scope of this work and solutions to the current limitations presented. First, the main limitation of this work is the fact that the models (both VNE and math-heuristic) cannot have been executed in a scalable way (in terms of memory and processing). Therefore, changes can be made to both models to optimize them and make their execution scalable in the terms mentioned above.

In addition, this work addresses the mapping of in-network neural networks as an optimization problem, not addressing implementation issues and the impact of these neural networks in practice (for example, implemented on programmable devices). Therefore, it is possible to evaluate the influence of implementing the solutions generated in this work on a programmable network device or smartNIC, aiming, for example, to assess the impact that the generated mappings have on the performance of the devices.

Bibliography

ABBASLOO, S.; YEN, C.-Y.; CHAO, H. J. Classic meets modern: a pragmatic learning-based congestion control for the internet. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 632–647. Cited 2 times in the pages 40 and 46.

AMAZON. **Machine Learning on AWS**. 2020. Disponível em: <https://aws.amazon.com/machinelearning/>. Cited in page 42.

AULD, T.; MOORE, A. W.; GULL, S. F. Bayesian neural networks for internet traffic classification. **IEEE Transactions on neural networks**, IEEE, v. 18, n. 1, p. 223–239, 2007. Cited in page 37.

BAI, J. et al. Fastfe: Accelerating ml-based traffic analysis with programmable switches. In: **Proceedings of the Workshop on Secure Programmable Network Infrastructure**. [S.l.: s.n.], 2020. p. 1–7. Cited 2 times in the pages 40 and 46.

BARMAN, D.; MATTA, I. Model-based loss inference by tcp over heterogeneous networks. In: **Proceedings of WiOpt**. [S.l.: s.n.], 2004. p. 364–73. Cited in page 37.

BASAT, R. B. et al. Constant time updates in hierarchical heavy hitters. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 127–140. ISBN 978-1-4503-4653-5. Cited in page 23.

BENSON, T.; AKELLA, A.; MALTZ, D. A. Unraveling the complexity of network management. In: **NSDI**. [S.l.: s.n.], 2009. p. 335–348. Cited in page 27.

BORGELT, C.; KRUSE, R. Induction of association rules: Apriori implementation. In: SPRINGER. **Compstat**. [S.l.], 2002. p. 395–400. Cited in page 37.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM 14**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Cited in page 23.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 3, p. 87–95, 2014. Cited in page 31.

BOUTABA, R. et al. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. **Journal of Internet Services and Applications**, Springer, v. 9, n. 1, p. 1–99, 2018. Cited 2 times in the pages 33 and 42.

BRAGA, A. d. P. **Redes neurais artificiais: teoria e aplicações**. [S.l.]: Livros Técnicos e Científicos, 2000. Cited in page 34.

CASE M. FEDOR, M. S. C. D. J. **Simple Network Management Protocol (SNMP)**. [S.l.], 1989. Disponível em: <https://www.hjp.at/doc/rfc/rfc1098.txt>. Cited in page 32.

CASTRO, A. G. et al. Patcher: Towards fault-tolerant probing planning for in-band network telemetry. In: IEEE. **2020 IEEE Latin-American Conference on Communications (LATINCOM)**. [S.l.], 2020. p. 1–6. Cited in page 74.

CASTRO, A. G. de et al. Análise do Desempenho de Heurísticas na Coleta de Informações de Telemetria In-Band. In: **17a Escola Regional de Redes de Computadores**. Alegrete-RS, Brasil: [s.n.], 2019. Disponível em: <http://errc.sbc.org.br/2019/papers/castro2019anlise.pdf>. Cited in page 74.

CHENG, Y. Mean shift, mode seeking, and clustering. **IEEE transactions on pattern analysis and machine intelligence**, IEEE, v. 17, n. 8, p. 790–799, 1995. Cited in page 37.

CISCO. **Cisco Visual Networking Index (VNI) Complete Forecast Update, 2017 - 2022. Technical Report. Cisco Systems, Inc.** 2020. [Retrieved: October 26, 2020]. Disponível em: <https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1213-business-services-ckn.pdf>. Cited in page 38.

CONSORTIUM, P. L. et al. P4 runtime. **Website, https://github. com/p4lang/PI**, 2017. Cited in page 31.

DASARI, M. et al. Streaming 360-degree videos using super-resolution. In: IEEE. **IEEE INFOCOM 2020-IEEE Conference on Computer Communications**. [S.l.], 2020. p. 1977–1986. Cited in page 38.

DEFAYS, D. An efficient algorithm for a complete link method. **The Computer Journal**, Oxford University Press, v. 20, n. 4, p. 364–366, 1977. Cited in page 37.

DU, K. et al. Server-driven video streaming for deep learning inference. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 557–570. Cited 2 times in the pages 39 and 46.

ERMAN, J. et al. Identifying and discriminating between web and peer-to-peer traffic in the network core. In: **Proceedings of the 16th international conference on World Wide Web**. [S.l.: s.n.], 2007. p. 883–892. Cited in page 37.

ESTRADA-SOLANO, F.; CAICEDO, O. M.; FONSECA, N. L. D. Nelly: Flow detection using incremental learning at the server side of sdn-based data centers. **IEEE Transactions on Industrial Informatics**, IEEE, v. 16, n. 2, p. 1362–1372, 2019. Cited 2 times in the pages 39 and 46.

FISCHER, A. et al. Virtual network embedding: A survey. **IEEE Communications Surveys & Tutorials**, IEEE, v. 15, n. 4, p. 1888–1906, 2013. Cited 3 times in the pages 49, 60, and 73.

FOSTER, N. et al. Frenetic: A high-level language for openflow networks. In: **Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow**. New York, NY, USA: ACM, 2010. (PRESTO '10), p. 6:1–6:6. ISBN 978-1-4503-0467-2. Disponível em: <http://doi.acm.org/10.1145/1921151.1921160>. Cited in page 30.

FOULADI, R. F.; SEIFPOOR, T.; ANARIM, E. Frequency characteristics of dos and ddos attacks. In: IEEE. **2013 21st Signal Processing and Communications Applications Conference (SIU)**. [S.l.], 2013. p. 1–4. Cited in page 24.

FUNDATION, O. N. **OpenFlow Switch Specification**. 2009. [Online; accessed 03-December-2020]. Disponível em: <https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf>. Cited in page 29.

GAJJALA, R. R. et al. Huffman coding based encoding techniques for fast distributed deep learning. In: **Proceedings of the 1st Workshop on Distributed Machine Learning**. New York, NY, USA: Association for Computing Machinery, 2020. (DistributedML'20), p. 21–27. ISBN 9781450381826. Disponível em: <https://doi.org/10.1145/3426745.3431334>. Cited 2 times in the pages 41 and 46.

GAO, J. et al. Scouts: Improving the diagnosis process through domain-customized incident routing. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 253–269. Cited 2 times in the pages 39 and 46.

GARG, P.; WANG, Y. Nvgre: Network virtualization using generic routing encapsulation. **RFC 7637**, 2015. Cited in page 31.

GEURTS, P.; KHAYAT, I. E.; LEDUC, G. A machine learning approach to improve congestion control over wireless computer networks. In: IEEE. **Fourth IEEE International Conference on Data Mining (ICDM'04)**. [S.l.], 2004. p. 383–386. Cited in page 37.

GOOGLE. **AI Platform**. 2020. Disponível em: <https://cloud.google.com/ai-platform/>. Cited in page 42.

GOWER, J. C.; ROSS, G. J. Minimum spanning trees and single linkage cluster analysis. **Journal of the Royal Statistical Society: Series C (Applied Statistics)**, Wiley Online Library, v. 18, n. 1, p. 54–64, 1969. Cited in page 37.

GUAN, T. et al. Recursive binary neural network training model for efficient usage of on-chip memory. **IEEE Transactions on Circuits and Systems I: Regular Papers**, IEEE, v. 66, n. 7, p. 2593–2605, 2019. Cited 3 times in the pages 44, 45, and 46.

HAMDAN, M. et al. Flow-aware elephant flow detection for software-defined networks. **IEEE Access**, IEEE, v. 8, p. 72585–72597, 2020. Cited 2 times in the pages 42 and 46.

HARIRI, B.; SADATI, N. Nn-red: an aqm mechanism based on neural networks. **Electronics Letters**, IET, v. 43, n. 19, p. 1053–1055, 2007. Cited in page 37.

HIDBER, C. Online association rule mining. **ACM Sigmod Record**, ACM New York, NY, USA, v. 28, n. 2, p. 145–156, 1999. Cited in page 37.

HOFMANN, T. Probabilistic latent semantic analysis. **arXiv preprint arXiv:1301.6705**, 2013. Cited in page 37.

HOHEMBERGER, R. et al. Orchestrating in-band data plane telemetry with machine learning. **IEEE Communications Letters**, IEEE, v. 23, n. 12, p. 2247–2251, 2019. Cited 4 times in the pages 23, 33, 50, and 74.

JAIN, S. et al. B4: Experience with a globally-deployed software defined wan. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 43, n. 4, p. 3–14, 2013. Cited in page 30.

JIANG, S. et al. A clustering-based method for unsupervised intrusion detections. **Pattern Recognition Letters**, Elsevier, v. 27, n. 7, p. 802–810, 2006. Cited in page 37.

KAYACIK, H. G.; ZINCIR-HEYWOOD, A. N.; HEYWOOD, M. I. On the capability of an som based intrusion detection system. In: IEEE. **Proceedings of the International Joint Conference on Neural Networks, 2003.** [S.l.], 2003. v. 3, p. 1808–1813. Cited in page 37.

Kim, H.; Feamster, N. Improving network management with software defined networking. **IEEE Communications Magazine**, v. 51, n. 2, p. 114–119, 2013. Cited in page 27.

KIM, J. et al. Neural-enhanced live streaming: Improving live video ingest via online learning. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 107–125. Cited 2 times in the pages 38 and 46.

KONEčNý, J. et al. Federated learning: Strategies for improving communication efficiency. In: **NIPS Workshop on Private Multi-Party Machine Learning**. [s.n.], 2016. Disponível em: <https://arxiv.org/abs/1610.05492>. Cited in page 42.

KOPONEN, T. et al. Onix: A distributed control platform for large-scale production networks. In: **OSDI**. [S.l.: s.n.], 2010. v. 10, p. 1–6. Cited in page 30.

KOURTELLIS, N.; KATEVAS, K.; PERINO, D. Flaas. **Proceedings of the 1st Workshop on Distributed Machine Learning**, ACM, Dec 2020. Disponível em: <http://dx.doi.org/10.1145/3426745.3431337>. Cited 2 times in the pages 42 and 46.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219. Cited 2 times in the pages 15 and 30.

LANDAUER, T. K.; DUMAIS, S. T. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. **Psychological review**, American Psychological Association, v. 104, n. 2, p. 211, 1997. Cited in page 37.

LEE, R.; VENIERIS, S. I.; LANE, N. D. **Neural Enhancement in Content Delivery Systems: The State-of-the-Art and Future Directions**. 2020. Cited 2 times in the pages 38 and 46.

LI, Y. et al. Inter-data-center network traffic prediction with elephant flows. In: IEEE. **NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium**. [S.l.], 2016. p. 206–213. Cited in page 37.

LI, Y. et al. Accelerating distributed reinforcement learning with in-switch computing. In: IEEE. **2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.], 2019. p. 279–291. Cited 3 times in the pages 44, 45, and 46.

LI, Y. et al. Reducto: On-camera filtering for resource-efficient real-time video analytics. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 359–376. Cited 2 times in the pages 39 and 46.

LI, Y. et al. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In: IEEE. **2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2018. p. 175–188. Cited 2 times in the pages 44 and 46.

LIKAS, A.; VLASSIS, N.; VERBEEK, J. J. The global k-means clustering algorithm. **Pattern recognition**, Elsevier, v. 36, n. 2, p. 451–461, 2003. Cited in page 37.

LIU, J.; MATTA, I.; CROVELLA, M. End-to-end inference of loss nature in a hybrid wired/wireless environment. In: . [S.l.: s.n.], 2003. Cited in page 37.

LIU, Y.; LI, W.; LI, Y. Network traffic classification using k-means clustering. In: IEEE. **Second international multi-symposiums on computer and computational sciences (IMSCCS 2007)**. [S.l.], 2007. p. 360–365. Cited in page 37.

LIU, Z. et al. Netvision: Towards network telemetry as a service. In: **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2018. p. 247–248. ISSN 1092-1648. Cited in page 32.

MAATEN, L. v. d.; HINTON, G. Visualizing data using t-sne. **Journal of machine learning research**, v. 9, n. Nov, p. 2579–2605, 2008. Cited in page 37.

MAHALINGAM, M. et al. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. **RFC**, v. 7348, p. 1–22, 2014. Cited in page 31.

MCDANEL, B.; TEERAPITTAYANON, S.; KUNG, H. Embedded binarized neural networks. **arXiv preprint arXiv:1709.02260**, 2017. Cited in page 42.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 38, n. 2, p. 69–74, 2008. Cited in page 28.

MEISTER, M. et al. Maggy: Scalable asynchronous parallel hyperparameter search. In: **Proceedings of the 1st Workshop on Distributed Machine Learning**. New York, NY, USA: Association for Computing Machinery, 2020. (DistributedML'20), p. 28–33. ISBN 9781450381826. Disponível em: <https://doi.org/10.1145/3426745.3431338>. Cited 2 times in the pages 41 and 46.

MENG, Z. et al. Interpreting deep learning-based networking systems. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 154–171. Cited 2 times in the pages 41 and 46.

MIJUMBI, R. et al. Network function virtualization: State-of-the-art and research challenges. **IEEE Communications surveys & tutorials**, IEEE, v. 18, n. 1, p. 236–262, 2015. Cited in page 30.

MIRSKY, Y. et al. Kitsune: an ensemble of autoencoders for online network intrusion detection. **arXiv preprint arXiv:1802.09089**, 2018. Cited in page 40.

MITCHELL, T. M. et al. Machine learning. McGraw-hill New York, 1997. Cited in page 33.

MORADI, M.; ZULKERNINE, M. A neural network based system for intrusion detection and classification of attacks. In: IEEE LUX-EMBOURG-KIRCHBERG, LUXEMBOURG. **Proceedings of the IEEE international conference on advances in intelligent systems-theory and applications**. [S.l.], 2004. p. 15–18. Cited in page 37.

NASR, M.; BAHRAMALI, A.; HOUMANSADR, A. Deepcorr: Strong flow correlation attacks on tor using deep learning. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.: s.n.], 2018. p. 1962–1976. Cited in page 40.

ONF. **Open networking foundation**. 2014. Disponível em: <https://www.opennetworking.org/>. Cited 2 times in the pages 28 and 30.

PAN, T. et al. Int-path: Towards optimal path planning for in-band network-wide telemetry. In: IEEE. **IEEE INFOCOM 2019-IEEE Conference on Computer Communications**. [S.l.], 2019. p. 487–495. Cited in page 33.

PAN, Z.-S. et al. Hybrid neural network and c4. 5 for misuse detection. In: IEEE. **Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 03EX693)**. [S.l.], 2003. v. 4, p. 2463–2467. Cited in page 37.

PETRINI, F.; VANNESCHI, M. k-ary n-trees: High performance networks for massively parallel architectures. In: IEEE. **Proceedings 11th international parallel processing symposium**. [S.l.], 1997. p. 87–93. Cited in page 59.

Pizzutti, M.; Schaeffer-Filho, A. Adaptive multipath routing based on hybrid data and control plane operation. In: **IEEE Conference on Computer Communications (INFOCOM 2019)**. Paris, France: IEEE, 2019. p. 730–738. ISSN 0743-166X. Cited in page 23.

QIN, Q. et al. Line-speed and scalable intrusion detection at the network edge via federated learning. In: IEEE. **2020 IFIP Networking Conference (Networking)**. [S.l.], 2020. p. 352–360. Cited 2 times in the pages 44 and 46.

RASHELBACH, A.; ROTTENSTREICH, O.; SILBERSTEIN, M. A computational approach to packet classification. **arXiv preprint arXiv:2002.07584**, 2020. Cited 2 times in the pages 40 and 46.

REICH, J. et al. Modular sdn programming with pyretic. **Technical Reprot of USENIX**, 2013. Cited in page 30.

SANVITO, G. S. D.; BIFULCO, R. Can the network be the ai accelerator? In: **Proceedings of the 2018 Workshop on In-Network Computing**. [S.l.: s.n.], 2018. (Workshop on In-Network Computing), p. 20–25. Cited 5 times in the pages 24, 44, 45, 46, and 73.

SAPIO, A. et al. In-network computation is a dumb idea whose time has come. In: **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2017. p. 150–156. Cited 2 times in the pages 43 and 46.

Saquetti, M. et al. P4vbox: Enabling p4-based switch virtualization. **IEEE Communications Letters**, v. 24, n. 1, p. 146–149, Jan 2020. ISSN 2373-7891. Cited 2 times in the pages 47 and 73.

SCHUBERT, E. et al. Dbscan revisited, revisited: why and how you should (still) use dbscan. **ACM Transactions on Database Systems (TODS)**, ACM New York, NY, USA, v. 42, n. 3, p. 1–21, 2017. Cited in page 37.

SIRACUSANO, G.; BIFULCO, R. In-network neural networks. In: **arXiv preprint arXiv:1801.05731**. [S.l.: s.n.], 2018. Cited 4 times in the pages 24, 43, 45, and 46.

SIRACUSANO, G. et al. Running neural networks on the nic. **arXiv preprint arXiv:2009.02353**, 2020. Cited 2 times in the pages 43 and 46.

SIRACUSANO, G. et al. Deep learning inference on commodity network interface cards. 2018. Cited 3 times in the pages 43, 45, and 46.

SUN, R. et al. Traffic classification using probabilistic neural networks. In: IEEE. **2010 Sixth International Conference on Natural Computation**. [S.l.], 2010. v. 4, p. 1914–1919. Cited in page 37.

WU, P. et al. Transition from ipv4 to ipv6: A state-of-the-art survey. **IEEE Communications Surveys & Tutorials**, IEEE, v. 15, n. 3, p. 1407–1424, 2012. Cited in page 27.

XIONG, Z.; ZILBERMAN, N. Do switches dream of machine learning? toward in-network classification. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2019. (HotNets '19), p. 25–33. ISBN 9781450370202. Cited 4 times in the pages 23, 24, 43, and 46.

XIONG, Z.; ZILBERMAN, N. Do switches dream of machine learning? toward in-network classification. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2019. p. 25–33. Cited in page 45.

ZHU, Y.; ZHANG, G.; QIU, J. Network traffic prediction based on particle swarm bp neural network. **J. Networks**, v. 8, n. 11, p. 2685–2691, 2013. Cited in page 37.

Index