

Distributed Graph Neural Networks in Programmable Data Planes

Ivan P. Lamb*, Pedro Arthur P. R. Duarte*, Jonatas Marques*, Marcelo C. Luizelli[†], Luciano P. Gaspar^y*, Anderson Tavares*, Ronaldo A. Ferreira[‡], Ítalo Cunha[§], José Rodrigo F. Azambuja*, Weverton Cordeiro*

*Federal University of Rio Grande do Sul (UFRGS) - PPGC, [†]Federal University of the Pampas (UNIPAMPA),

[‡]Federal University of Mato Grosso do Sul (UFMS), [§]Federal University of Minas Gerais (UFMG)

*{first.last}@inf.ufrgs.br, [†]marceloluizelli@unipampa.edu.br, [‡]raf@facom.ufms.br, [§]cunha@dcc.ufmg.br

Abstract—The ability to redefine the data plane behavior with programmable network devices provides a plethora of novel possibilities for in-network computing. One of these possibilities is embedding Artificial Intelligence (AI) and Machine Learning (ML) techniques directly in the data plane. Motivations include reducing decision latency and closing the control loop—*i.e.*, performing measurements, learning, decisions, and actions directly in the data plane. However, running entire AI/ML algorithms in a single device might be infeasible due to memory and computing constraints. This work addresses the research challenges of running a Graph Neural Network (GNN) in a set of devices of a programmable data plane. Our hypothesis is that by distributing the GNN processing across the devices, the GNN uses instantaneous snapshots of the global network state and can act more quickly. As a proof of concept, we trained and evaluated a distributed GNN to perform explicit congestion notifications based on Data Center Transmission Control Protocol (DCTCP). We verified the feasibility of GNN classification in the data plane through simulations and both software and hardware switch experiments with bmv2 and Intel Tofino.

Index Terms—Graph Neural Networks, Programmable Data Planes, DCTCP

I. INTRODUCTION

Software Defined Networking (SDN) has eased management, supported more complex policies, and allowed finer traffic control in several contexts, including enterprises, Internet exchange points, wide-area networks, and content distribution networks [16], [17], [30], [9]. More recently, Programmable Data Planes (PDP) have allowed offloading packet processing functionalities from network middleboxes to the data plane [4]. In this context, many high-performance in-network solutions employing PDPs have been proposed for applications such as monitoring, security, storage, and DNS processing [33], [31], [11]. This strategy reduces the workload on commodity servers and end hosts, leveraging PDP line-rate hardware processing power. More importantly, the delays in communication between the data and control planes are removed.

In a similar trend, Artificial Intelligence (AI) and Machine Learning (ML) revolutionized several fields, enabling the implementation of complex functions learned from data [3]. In the context of network communications, AI and ML could be fundamental building blocks of self-driving networks [12], [20], supporting solutions for tasks such as congestion control,

traffic engineering, or anomaly detection [32], [2]. Combining AI and ML with SDN and PDPs to run novel solutions directly on the data plane could achieve unprecedented fine decision granularity and low response times.

In this context, many computer networking concepts (just like many other scientific domains) can be intuitively represented as graphs. Specifically in networking, those concepts range from relationships between users of a Local Area Network (LAN) and their permissions to the very topology of the networks. By noticing those numerous applications of graphs and the ever-growing application of ANNs in numerous fields, the Graph Neural Network (GNN) model [29] was proposed to perform classification with input data specifically structured as graphs, with the main motivation being the poor performance of the other consolidated neural network models when dealing with graph-structured data.

PDP devices, in turn, can derive/extract inputs and metrics like queue occupation, flow rate, and round-trip delay and use the network to enable neighboring nodes to exchange them in a GNN having a topology similar to/equal to that of the network itself. Additionally, with PDPs, it is possible to take advantage of currently deployed hardware without requiring network operators to acquire new GNN-enabled forwarding devices because they are reconfigurable.

Existing research on GNNs applied to networking has either focused on modeling network behavior [14] or determining network operations & management actions in the control plane [26]. However, the ultra-low-latency packet processing in switches and ever-increasing link speeds make it challenging for AI/ML applications to keep up with network management decisions on the order of hundreds of nanoseconds.

In this paper, we emphasize that switches, being strategically distributed throughout the network core, are especially effective at gathering state information from multiple links. Leveraging this advantage, we propose implementing and running Graph Neural Networks (GNNs) directly and in a distributed manner on Programmable Data Planes (PDPs) [36]. Throughout the paper, we enumerate several challenges we envisage in relation to the training and classification of GNNs. While we point out that training GNNs in PDPs nowadays is essentially impractical, we provide solid insights on over-

coming classification challenges that ultimately lead to the conclusion that it is both feasible and practical with today's hardware. We also present a framework to perform GNN classification with a use case for controlling congestion on data center networks based on Data Center Transmission Control Protocol (DCTCP), which we use to provide evidence that hyperparameters used for training the GNNs have a significant impact on accuracy. To the best of our knowledge, our work is the first to present a preliminary discussion on GNNs running distributedly directly in PDPs by (i) identifying research challenges that may be central to enabling successful GNN solutions and deployments, (ii) presenting and dissecting results obtained in a simulated environment and (iii) providing a framework to generate GNN implementations in the form of data plane code given user-defined GNN models. The source code, data, and scripts produced in our research are available so that researchers can reproduce our experiments and verify our findings.

The remainder of this paper is organized as follows. Section II provides a brief background on PDPs and GNNs. Section III discusses the requirements and challenges of integrating GNN classification with the PDP architecture. Section IV provides simulation results of GNN classification tasks executing in the core of a network. Section V describes our framework that generates GNN code for real PDP devices. Finally, Section VI closes the paper with concluding remarks and prospective directions for future work.

II. BACKGROUND

In this section we describe concepts and technologies that constitute the building blocks for our proposed approach.

A. Programmable Data Planes

SDN emerged as a result of research efforts to promote innovation in networking [13]. The main contribution of SDN was the separation of control and data planes, enabling both planes to evolve independently while communicating through a standardized protocol, such as OpenFlow [21]. The SDN architecture divides the network into three distinct planes:

- **Application Plane:** Consists of applications that implement a specific network functionality. Operators can deploy multiple applications, such as a routing application, a firewall application, a monitoring application, etc. These applications communicate only with the control plane through the Northbound Interface.
- **Control Plane:** A centralized entity¹ that translates the requests from the application plane to the devices in the plane. It may also provide other functionalities, such as authentication and security. It provides the applications with an abstract view of the forwarding elements (switches, routers, etc.) in the data plane.
- **Data Plane:** This is where the forwarding elements are located. They receive the forwarding logic from the

control plane through the Southbound Interface, also known as the Control-Data-Plane Interface (CDPI). One of the advantages of the SDN architecture is that the data plane devices can originate from different vendors and even be implemented with various technologies, such as FPGAs, ASICs, software emulation, and others. The CDPI standardization enables these different elements to be deployed in the same environment.

Due to the possibility of executing in commodity servers, the application and control planes saw great innovation due to the flexibility that general-purpose CPUs provide to operators. However, innovation had been hindered in the data plane due to forwarding devices being generally configurable but not strictly programmable [4]. In other words, OpenFlow is constrained to adding and removing entries from pre-defined tables supporting only a limited set of standardized protocols (e.g., Ethernet, IP, TCP). At the time OpenFlow was envisioned, network design was solely bottom-up, i.e., starting from the limited set of protocols supported by the data plane hardware and then seeing what could be built on top of these protocols to implement the applications and services demanded by the users.

When the concept of SDN emerged, the main goal was to provide network operators with more flexible tools for traffic engineering and monitoring. The driving force that made this possible was OpenFlow [21], which specifies an open interface to perform basic configurations on a network device on the go, such as adding and removing entries from a table. Even so, the OpenFlow specification limits the traffic the devices can process, such as well-known and standardized internet protocol headers (e.g., IP, TCP, Ethernet). To further provide innovation flexibility, the network community proposed ways to (re)define not only table entries and register values but the whole internal structure of each device.

Programmable Data Planes (PDPs) were introduced to bring greater flexibility to defining how forwarding devices should parse and process network packets. The main idea is first to develop packet-processing hardware independent of specific protocols [5] and then use domain-specific languages (e.g., P4 [4], NPL [6]) to program them. These languages offer high-level abstractions to specify packet processing pipelines, including custom protocol headers, parsing logic, stateful registers and counters, and match-action tables. P4, in particular, has had wide market adoption, with several commercial products available, including Intel Tofino, NetFPGA-SUME (Xilinx), and SmartNICs (Netronome and NVIDIA BlueField).

PDPs enable a top-down approach to network design, starting from application and service demands and moving toward custom algorithms and protocol implementations in the data plane. Among the novel opportunities PDPs bring is the ability to perform in-network computing. Our goal is to take advantage of this opportunity to design and implement an architecture for GNN processing that runs in the data plane.

¹The controller can be viewed as a logically centralized entity, but its actual implementation may be distributed [23].

B. Graph Neural Networks

A Graph Neural Network (GNN) is a model of Artificial Neural Networks (ANN) that handles graph-structured data [29]. This model overcomes the shortcomings of converting graphs to tabular data to be consumed by traditional ANN models. GNNs can generalize the trained models to different graph topologies, attracting the attention of researchers working on many different areas². Many research problems in computer networks have properties or structures (the network's topology is often represented in graph form) that make them suitable for modeling as graph problems. Therefore, GNNs are a popular research tool for ML studies in the computer networks area with a variety of applications such as intrusion detection [25] and performance estimation [26].

A classic architecture for GNNs is the Message Passing Neural Network (MPNN) [29], [15], in which the nodes and edges of the graph have “hidden state” vectors. These vectors encode a useful representation for the purpose of the model, being processed through the operations below:

- **Message:** A GNN iteration starts with each node producing a message to its neighbors. A *message* operation is modeled by a function that may take the hidden state of either the sender/receiver node or the edge connecting them as input.
- **Aggregation:** After receiving all the messages from its neighbors, each node produces a resulting vector with a function that aggregates the received messages. Typically, the aggregation function involves an average of the received messages.
- **Update:** Each node updates its hidden state with a function that takes as inputs its current hidden state and the produced aggregation vector.

After a defined number of iterations of these three operations, a readout operation extracts the useful information for each node, edge, or entire graph. The application then uses this information as it would use the result of a traditional calculation. Typically, the functions for update and readout operations are implemented with trainable ANN models, such as Multi-Layer Perceptrons (MLPs) or recurrent neural networks. The parameters of the trainable components of the GNN are adjusted via gradient descent on a loss function defined over the difference between GNN predictions and target values from the dataset.

III. REQUIREMENTS AND CHALLENGES TOWARDS GNNs ON PROGRAMMABLE DATA PLANES

One must address many requirements and challenges to run GNNs directly (and distributedly) in PDPs. These requirements and concerns involve both the training and deployment of the GNN and its behavior when considering each *message*, *aggregation*, *update*, and *readout* operations on the hidden state vectors. Next, we discuss these requirements and challenges, considering a distributed deployment environment of the GNN in the core network devices (*switches*).

In a PDP environment, switches have limited memory and computing resources and observe a subset of traffic traversing the network. These constraints may become a limitation during a *message* exchange operation when each switch sends in each iteration its internal hidden state (feature vector) to the neighboring switches (*i.e.*, these switches to which it has a direct link). During the *aggregation* phase, the switches aggregate the received messages. Finally, the switch uses the resulting aggregation to update its internal hidden state vector during the *update* phase. All of these steps consume stages in the pipeline of programmable devices and must be fit accordingly in addition to the application logic that ultimately calls the classification services of the GNN.

A. Requirement #1: GNN Classification

Our approach for running a GNN distributedly in a programmable data plane relies on per-switch processing: each switch in the network that is part of the GNN hosts a single GNN node (*i.e.*, there is a one-to-one mapping from the nodes to the switches). Recall that, considering that a GNN may take the form of the graph data it represents, a distributed GNN might assume the topology of the underlying network.

Challenge 1. Resource limitations complicate node placement, as switches may not have enough resources to host a particular GNN node, especially if the use-case application already consumes much of the target's available resources. Addressing this may require rebuilding the GNN and/or remapping nodes onto different switches. Possible approaches for node placement may involve combining an abstract description of the GNN with topological information of the PDP as inputs to a compiler, which will simultaneously split the GNN into nodes that can be mapped onto switches. We expect solutions to Virtual Network Function mapping may serve as starting points [28]. On that note, GNNs whose input graphs are the same as the network topology graph and are light regarding data plane resource usage are possibly the best suited for PDP classification, such as RouteNet latency estimation [26].

B. Requirement #2: GNN Communication

On each iteration of a distributed GNN, each switch generates and sends messages to other neighboring switches, aggregates messages received from neighboring switches, and updates its hidden state (§II). Messages between nodes could be embedded opportunistically in network traffic (*e.g.*, in an additional packet header) or sent in new packets for this specific purpose. Each approach comes with trade-offs: embedding messages opportunistically may increase packet size and lead to fragmentation; it could also impose additional requirements on switch hardware to modify packets in transit. Generating additional packets incurs bandwidth overhead and must be done either sporadically (which many hardware targets do not support) or by cloning network traffic packets.

Challenge 2. The frequency at which messages are exchanged determines resource utilization on switches, bandwidth overhead, and GNN responsiveness. More importantly, the frequency of message exchanges also determines the level of

²A comprehensive survey on GNNs is provided by [34].

synchronization among nodes, impacting the accuracy of the GNN and the effectiveness of training [10]. Deciding the frequency at which messages are exchanged is an important metaparameter. In §IV-A, we present an evaluation of this meta-parameter and discuss its impact.

Challenge 3. At the intersection of Requirements 1 and 2 lie several configuration and implementation decisions that will impact the deployment and efficacy of GNNs on data planes. For example, what data should be shared among nodes? Should a switch share information about all its ports with each neighbor or a subset of ports? Should GNN messages be propagated only 1-hop or multiple hops? In general, insufficient information may lead to inaccurate models. In contrast, excessive information will quickly become intractable and are more prone to overfitting. Researchers must identify these restrictions and build frameworks to evaluate their impact, adjusting accordingly to the available network resources.

C. Requirement #3: GNN Training

Large volumes of traffic in today’s networks may serve as input data for training GNNs. However, training also requires labeled inputs when using supervised learning or computing rewards for GNN actions when using reinforcement learning.

Challenge 4. Unfortunately, GNN may alter network traffic (e.g., add delays or reroute flows) and thus any existing labels attached to it, making classical supervised learning approaches unfit. When using unsupervised learning, computing rewards based on network properties such as link utilization and queue occupation should be straightforward, as this information is readily available from switches. However, an increasing fraction of applications depend on complex, higher-level metrics, such as buffering time and the number of stalls in video streaming. Solutions to this challenge may involve developing training frameworks that operate over intervals long enough to compute and integrate metrics from the application layer.

Challenge 5. Programmable data plane architectures often lack native support for floating-point arithmetic, thus hampering artificial neuron implementation. The research community has worked around this limitation in various contexts, either using fixed-point arithmetic [28] or decomposing floating-point functions into lookup tables [18]. The trade-off between precision and resource usage is an aspect to consider in a real-world GNN deployment.

IV. EXPERIMENTAL EVALUATION THROUGH SIMULATION

In this paper, we aim to assess the feasibility of GNN classification in PDPs. We developed an algorithm to be deployed in the network core devices to perform the steps described in Section II-B. We employed Network Simulator 3 (NS3) [22], a popular simulator for computer network experiments, to implement a prototype version of the algorithm. It is a C++-based discrete event simulator powerful enough to simulate multiple scenarios with packet-level precision, ideal for prototyping novel protocols and network algorithms. The simulated experiments allowed us to fine-tune the algorithm

in a more flexible environment before proceeding to the actual PDP implementation described in Section V.

A. Case-Study: DCTCP ECN markings

To verify the feasibility of using GNNs to achieve specific goals and evaluate their efficacy, we perform a case study to evaluate whether GNNs can replicate the widely deployed Data Center Transmission Control Protocol (DCTCP) [1]. DCTCP was originally proposed to improve the throughput and delay of TCP traffic in a data center. Essentially, it is a modification of the classic TCP that leverages Explicit Congestion Notification (ECN) markings on packets so that the transmitting hosts can react faster to queue buildup in intermediate switches, as opposed to the classical TCP approach of detecting congestion by packet loss. More precisely, switches set the CE codepoint in packet headers when queue occupations are above a threshold. This threshold (and, more generally, the packet marking algorithm) is opaque to TCP implementations running on end-hosts and usually set to a small value to ensure low latencies.

We can model DCTCP’s packet marking as a GNN classification task. Given a snapshot of the network topology at a given time, we model each queue in a forwarding device as a GNN node with its hidden state initialized with queue occupation information. For the edges, there are multiple approaches to determine the neighborhood of a node (§III). Here, we create an edge from a GNN node representing a queue Q_1 to another GNN node representing queue Q_2 if packets leaving Q_1 may be queued at Q_2 . In other words, we connect nodes Q_1 and Q_2 if Q_1 ’s output port is connected to the switch owning Q_2 . Other possible neighborhoods may involve connections only between more directly related nodes, such as queues that packets visit in sequence, which leads to fewer edges in the GNN graph and may reduce communication overhead. This formulation illustrates one possible advantage of GNNs over classical approaches: We allow each queue to leverage information about the queue occupation of its neighbors in its marking decision, as opposed to DCTCP, where the decision depends only on local queue occupancy.

B. Dataset Generation and Model Training

Before implementing our GNN classification framework on the simulator, we collected a dataset to train a GNN to perform ECN markings for DCTCP based on local and neighboring queue occupations. We ran a simulation of a data center using a fat tree topology with eight TOR switches, each with four end hosts connected to them. Links between end hosts and switch have 1.0Gbps speed and 10 μ s latency. Links between switches have 1.5Gbps speed and 10 μ s latency.

We recorded the queue occupation for each queue in the network every 100 microseconds for 30 seconds. Each of the 32 hosts sends data sporadically to another host through a DCTCP connection throughout the experiment. This scenario was used for all experiments for performance comparison. This experiment generated a temporal dataset with 300,000 samples, each representing a snapshot of the network at a given

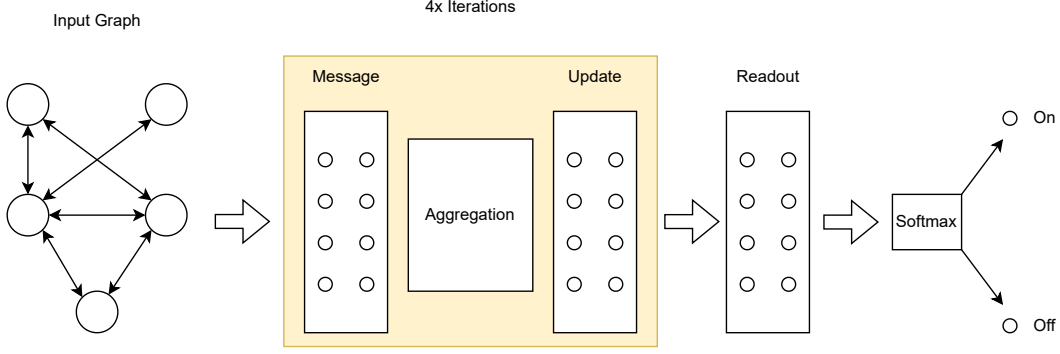


Fig. 1. GNN Experiment Model.

TABLE I
RESULTS FOR THE SIMULATED EXPERIMENTS

	No ECN	DCTCP	GNN (10ms)	GNN (1ms)	GNN (100us)
Average Throughput (MBit/s)	170.65	313.03	152.42	220.24	312.73
Average GNN Accuracy			0.9087	0.9399	0.9645
Average GNN Precision			0.1259	0.5158	0.9573
Average Message Exchange BW Overhead			6.82×10^{-5}	6.82×10^{-4}	6.82×10^{-3}

TABLE II
RESULTS FOR THE EXPERIMENT CONSIDERING RANDOM TOPOLOGIES

	No ECN	DCTCP	GNN (10ms)	GNN (1ms)	GNN (100us)
Average Throughput (Compared to DCTCP)	93.54%	100%	99.63%	99.63%	99.63%

time, which we reduced to approximately 38,000 graphs by considering only the ones in which at least one of the queues marked a packet with ECN.

Building a labeled dataset for training the GNN to mimic DCTPC’s ECN markings is rather straightforward since the decision is purely based on a threshold on the queue occupations. To further explore the extra possibilities GNNs bring to the table, we introduced a new feature in our dataset that contains information about the rate at which each queue is filling up, computed as the difference between the occupation at the current snapshot and the previous one. Our hypothesis is that knowing neighbors’ queue occupation trends will allow for a better-informed final decision. We also trained our model to try to predict ECNs some time steps ahead of the observed in the dataset, which we do by training the GNN to predict the ECN markings at snapshot t using queue occupation information at a previous snapshot. This procedure allows the GNN to react earlier to changes in the network traffic, which may further improve TCP performance.

After acquiring the dataset, we trained our MPNN GNN with the tensorflow-based GNN training kit iGNNition [24]. iGNNition was designed for fast prototyping with GNNs in network scenarios, including examples of well-known model descriptions, such as RouteNet [26]. Our GNN model (shown in Fig. 1) is a simple MPNN with four iterations of *Message* (MLP), *Aggregation* (Average), and *Update* (MLP) steps followed by a *Readout* (MLP) for each node. After the Readout,

a *Softmax* operation is applied to the results to transform them into a binary value. In our test case, we use this binary value to determine if incoming packets should be marked with ECN.

C. Implementation

We modified NS3’s Random Early Detection (RED) queues to implement our GNN by replacing RED with the GNN’s output to set ECN marks and turning off packet dropping. Since each queue is mapped directly to a GNN node in this test case, we also modified the RED queues to include hidden state information and other GNN metadata. Note that, in an actual data plane implementation, each forwarding device needs a data structure to keep track of the hidden states of each of its GNN nodes (one node per queue).

Every forwarding device sends information about the occupation from each of its queues to its neighbors at each time interval via a small packet. The exact size of each packet varies with each case, depending on the dimensions of the hidden state vectors and the number of nodes per switch, which may vary depending on use case. The packet contains an ethernet header and a special GNN header with that information. The programmable devices can identify this header by a particular value in the ethernet type field of the ethernet header.

The time interval is another configurable parameter that needs to be tuned so that the switches can update the GNN classification more quickly and react to network traffic. However, we must also consider that more frequent messages incur higher communication overhead. In our experiments,

each GNN message packet is 40 bytes long and we evaluate multiple time interval values.

Since NS3 allows the user to define any processing operations for the network devices to perform using C++ code, it is possible to implement the classification algorithms of the GNN directly in the simulator. This implementation mainly includes NN forwarding, consisting of several loops for NN layers, multiplications, divisions, and activation functions. These operations are not trivially implemented in actual programmable data planes and will be discussed further in Section V.

For the classification, the GNN configuration is loaded from iGNNition's trained output weights at the start of the simulation, as opposed to the data plane implementation, in which the configuration is described in the code. We provide the source code of our simulated experiments with helpful scripts to reproduce the results in the simulator.³

D. Results

Table I contains the main results of our simulations with the fat tree topology with multiple hosts attached to each switch. We vary the GNN message exchange period between 10 ms, 1 ms, and 100 μ s. The table shows the average throughput for all test cases, the GNN accuracy and precision⁴ for the experiments that uses GNNs, and the average bandwidth overhead of the message exchanges. Bandwidth overhead is given by the fraction of the 1.5Gbps link bandwidth used for GNN messages, according to Eq. (1), where NQ is the number of queues, NS is the number of switches, $MSGSize$ is the size of messages, and $MSGFreq$ is the frequency of messages. As the equation implies, the higher the link bandwidths the lesser the communication overhead. Results show that with a high enough message frequency, one can achieve a precision of more than 95% with a relatively low overhead.

$$\frac{(NQ/NS) \times MSGSize \times MSGFreq}{LinkBandwidth} \quad (1)$$

Table II shows the results for an additional test case with multiple (100) randomly generated network graphs. This brief test case was performed to show that the results can be generalized to different graph topologies. The table contains the average percent of the total throughput achieved by the traditional DCTCP algorithm for each method of ECN marking. The results show that in terms of throughput, for those topologies, the GNN performs almost as well as the traditional DCTCP algorithm, and both perform better than traditional TCP connections (without ECN).

V. P4 FRAMEWORK IN A REAL-WORLD PDP TESTBED

After validating our proposal in the simulator, our next goal was to prove the feasibility of the proposed algorithm with actual data plane forwarding devices. In such an environment, other considerations not present in the simulated environment

must be made, including network delays, packet losses, and, most importantly, the restrictions to the algorithm imposed by the PDP language (P4).

To tackle these modifications, we implemented a framework to automatically generate P4 code that performs the algorithm given a pre-trained GNN description. The framework source code and scripts for the reproducibility of the experiments described in this section are also made publicly available.⁵

The framework facilitates the user's job, as they only need to define the NNs that implement the message, aggregation, update, and readout functions. Still, as of now, the user must perform the training prior to the deployment in the programmable data plane, providing the weights of the NN's along with their architectural description.

After the generation of the P4 code, the user needs to insert the sources and sinks of data in the code. In our ECN example, those would be the queue occupancy and a boolean variable to decide the marking of packets, respectively. This way, we can replace traditional decision-making algorithms with GNN classification that runs entirely and distributedly in the PDP.

Various targets support the P4 language, but different architectures impose different constraints on the language. As such, the code that is generated from the framework is target-dependent. For example, the set of possible operations on registers in the Tofino Native Architecture (supported by Intel Tofino switches) differs from the set of operations of the v1model architecture (supported by the bmv2 prototyping software switch). Currently, our framework allows code generation for those targets (bmv2 and Tofino). The generated codes for both architectures contain several differences, mainly because of the way the architecture implements registers and restricts the number of memory accesses per processed packet.

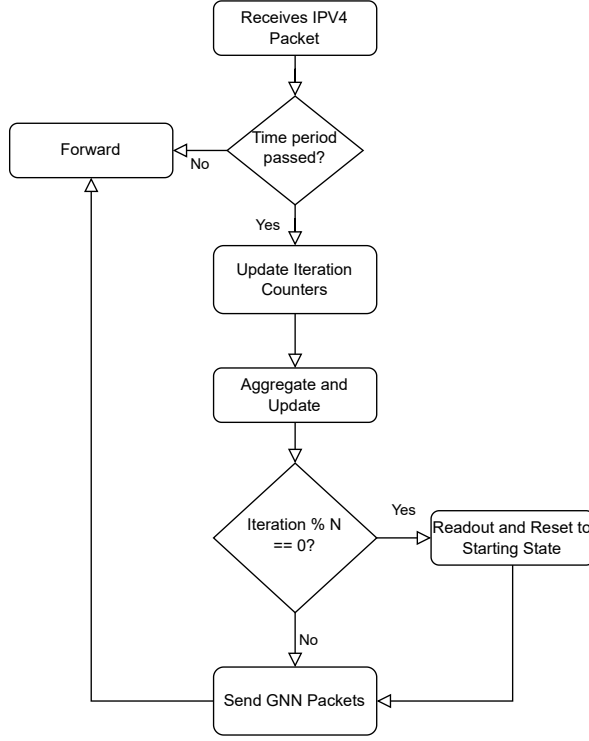
As mentioned in Section III, implementing generic algorithms such as NN processing in a PDP is more complex than implementing those algorithms in a general-purpose programming language. We list the main challenges that we encountered and the insights for the solutions:

- **Lack of floating point support:** We employ fixed point arithmetic for the multiplication operations. We also implement the necessary divisions when calculating averages of N neighbors with the multiplication by $1/N$. The number of bits employed for each part of the numbers (integer and fraction) is a hyperparameter that can impact the results of the classification. We found in our tests that 8 bits for the integer part and 24 for the fraction is a good compromise, considering the maximum of 32-bit values for multiplication supported by the Tofino hardware.
- **Creating packets:** We clone and modify ordinary traffic packets to perform GNN message exchange. After cloning the packets, we strip the L3 protocol (IPv4, in our tests) and insert our GNN protocol header in its place, containing the necessary information for each message. It is not necessary for each switch in the network to receive

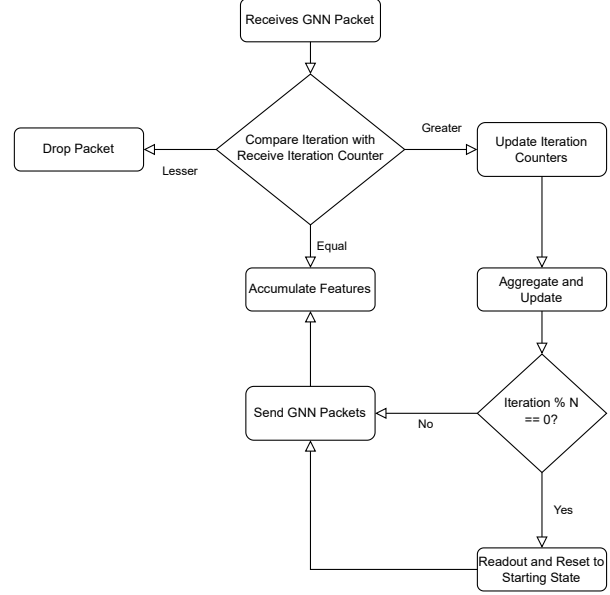
³GitHub link to the source code used in the simulations: <https://github.com/ComputerNetworks-UFRGS/NOMS2025-GNN-Dataplane>

⁴For this case, **precision** is a better metric than **accuracy** since the vast majority ($\approx 92\%$) of the instances should be labeled as negatives due to the imbalanced nature of ECN markings.

⁵GitHub link to the source code of our P4 framework: <https://github.com/ComputerNetworks-UFRGS/NOMS2025-GNN-Dataplane>



(a) IPv4 packets



(b) GNN packets

Fig. 2. Flowchart for the data plane GNN algorithm.

ordinary traffic for packet creation, as they can perform the same operation when receiving GNN packets from their neighbors. This way, a single packet received by any switch in the network is enough to trigger an iteration.

- **Loops:** We unroll each loop necessary for the multiple layers of NN processing. Since we are training the NNs before deploying it in the data plane program and sequentially apply each multiplication, aggregation, and activation function in the NN processing.

The modified algorithm that is implemented by the P4 framework is illustrated in Fig. 2. The iterations are triggered when a switch in the network receives a packet associated with traffic in the network. In the figure, the network processes IPv4 packets. Fig. 2(a) depicts this initial step, which will update internal variables and send GNN packets with the feature vectors when the period passes. Fig. 2(b) depicts the process that occurs when a switch receives such GNN packets, which depends on the iteration counter for the current GNN iteration. The iteration is important so that data from different iterations are not mixed, which may alter the classification result. Suppose the received packet has a counter lesser than the current iteration counter of the node. In this case, it means that the packet belongs to a previous iteration and has experienced delay due to events in the network, so it should be discarded. Packets with an equal iteration counter are accumulated in the feature vectors. Packets with a greater

counter indicate that a new iteration has begun, so the previous iteration is finalized (by applying the aggregate and update functions) before starting the next one with the received feature vector. When the number of iterations reaches N (defined by the user), the internal feature vector is reset, and the readout variables are updated for use by the application.

A. Evaluation

We evaluated the correctness of the algorithm with programmable software switches running on mininet. Additionally, we evaluated that a hardware-based Tofino switch can perform the same processing as the software switches.

Fig. 3 depicts the topology for the correctness experiment running on mininet. The network is comprised of three L3 switches in a triangle topology processing IP packets between hosts connected to each switch. The network is kept busy with iperf flows between each pair of hosts during the experiment, which alternates in throughput to generate periods of congestion. We employed the trained GNN classification model using the iGNNition framework described earlier in Section IV. We verified that the programmable switches were able to correctly perform the operations of all NNs defined by the model (message, update, and readout) and mark packets accordingly.

Fig. 4 depicts the topology for the DCTCP experiment. The topology includes two L3 switches with multiple hosts in their subnets. The switches are connected by a link, the traffic

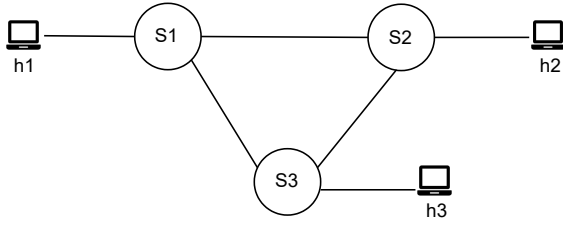


Fig. 3. Topology of the P4 correctness experiment.

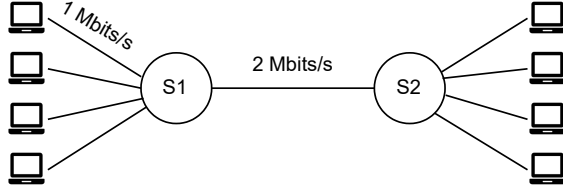


Fig. 4. Topology of the P4 DCTCP experiment.

TABLE III
TOTAL THROUGHPUT FOR THE TCP CONGESTION CONTROL SCHEMES

	Total Throughput (Mbits/s)	Jain Fairness
Cubic	1.929	0.83
DCTCP	2.161	0.94
GNN	2.144	0.95

bottleneck between hosts in different subnets. The experiment consists of 4 simultaneous iperf connections between pairs of hosts in each subnet, passing through the bottleneck link. The total duration of the experiment is 1 minute, and flows start with a delay of 10 seconds each. The topology is similar to the one used by DCTCP’s authors to evaluate the protocol [1]. Table III contains the results regarding the aggregated throughput of the iperf connections and the fairness of the traffic distribution based on the Jain index. The results are an average of 30 repetitions. The GNN results are similar to the traditional DCTCP algorithm and similar to the results achieved in the simulated environment. Both DCTCP and GNN algorithms perform better in terms of throughput and fairness than Cubic TCP, which does not rely on ECN.

The described experiments and the available scripts for reproducibility employed the bmv2 switch target. We also verified that the operations can be performed by Intel Tofino hardware switches, which are used in real data center applications, by employing a single hardware switch and traditional servers to mimic other devices in the network. The reproducibility of the experiments in full would require multiple programmable hardware switches, which we leave for further work.

VI. FINAL CONSIDERATIONS

In this work, we take the first steps toward leveraging PDP resources to assist in machine learning (ML) tasks in a distributed fashion. GNNs are a popular ML model, and their

applications in the area of computer networks are growing by the day. By performing GNN classification tasks directly in PDPs, we free up control plane resources in terms of processing time and eliminate the communication overhead that would occur between those planes, particularly as network speeds grow beyond 100G per link.

We presented an implementation of GNN logic in a simulator (without network delay considerations) to ascertain the potential applications of such a model in a specific use case involving ECN-enabled connections. We developed an algorithm to perform GNN classification in PDPs considering potential delays, packet losses, and many processing power restrictions of high-performance PDP switches. We implemented a framework to generate P4 code to implement the algorithm automatically. We tested the algorithm in both software switches, with multiple programmable devices in the network topology and a hardware switch (Intel Tofino) representing a node in the GNN. We verified that the experimental results with the PDP programs are similar to the ones achieved in the simulated environment. Results for the proposed test case, ECN markings, are similar to the traditional algorithm.

In future works, we intend to explore other use cases beyond congestion control to explore the opportunities offered by in-network GNNs. Research avenues also include assessing GNN classification tasks in switches that provide built-in machine learning features [35], [36]. For the long term, we intend to explore GNN models beyond the Message Passing Neural Network, including Graph Convolutional Networks, Graph Recurrent Networks and other models.

Additionally, we aim to ascertain the possibility of ML training directly in PDPs, which is much more challenging than classification, though more practical and convenient than offline training. One promising approach involves using unsupervised ML techniques, which have been successful in some applications running on SmartNICs [8]. Finally, another challenge worth attention is coupling AI/ML techniques with virtualization solutions for the data plane. There have been substantial investigations in the field and, although proposed solutions are maturing and enabling seamless deployment and operation of virtual switches [7], they still pose technical limitations in terms of available resources, switching performance, and constructs that can be used [27], [19].

ACKNOWLEDGEMENTS

This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and grant #88887.954253/2024-00. This study was also financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq) grants #314506/2023-3, #405940/2022-0, #404027/2021-0, São Paulo Research Foundation (FAPESP) grants 2020/05183-0 (Skynet), 2021/00199-8 (SMARTNESS), 2023/00673-7 (IDRCIC), 2023/00811-0, 2023/00812-7, and 2023/00816-2 and Rio Grande do Sul Research Foundation (FAPERGS) grant 24/2551-0001394-6.

REFERENCES

- [1] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). *SIGCOMM Comput. Commun. Rev.* 40, 4 (aug 2010), 63–74.
- [2] BERNÁRDEZ, G., SUÁREZ-VARELA, J., LÓPEZ, A., WU, B., XIAO, S., CHENG, X., BARLET-ROS, P., AND CABELLOS-APARICIO, A. Is Machine Learning Ready for Traffic Engineering Optimization? In *Proc. IEEE ICNP* (2021).
- [3] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [4] BOSSHART, P., DALY, D., GIBB, G., ET AL. P4: Programming Protocol-independent Packet Processors. In *Proc. ACM SIGCOMM* (2014).
- [5] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, Association for Computing Machinery, p. 99–110.
- [6] BROADCOM INC. Open, high-level language for developing feature-rich solutions for programmable networking platforms, 2022.
- [7] BUENO, G., SAQUETTI, M., RODRIGUES, P., LAMB, I., GASPARY, L., LUIZELLI, M. C., ZHANI, M. F., AZAMBUJA, J. R., AND CORDEIRO, W. Managing virtual programmable switches: Principles, requirements, and design directions. *IEEE Communications Magazine* 60, 2 (2022), 53–59.
- [8] CANNAROZZO, L., MORAIS, T. B., DE SOUZA, P. S. S., GOBETTO, L. R., LAMB, I. P., DUARTE, P. A. P. R., AZAMBUJA, J. R. F., LORENZON, A. F., ROSSI, F. D., CORDEIRO, W., AND LUIZELLI, M. C. Spinner: Enabling in-network flow clustering entirely in a programmable data plane. In *NOMS 2024-2024 IEEE Network Operations and Management Symposium* (2024), pp. 1–9.
- [9] CASADO, M., MCKEOWN, N., AND SHENKER, S. From ethane to sdn and beyond. *SIGCOMM Comput. Commun. Rev.* 49, 5 (nov 2019), 92–95.
- [10] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *Proc. USENIX ATC* (2014).
- [11] DANG, H. T. P4dns: In-network dns. *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2019).
- [12] FEAMSTER, N., AND REXFORD, J. Why (and how) networks should run themselves. *CoRR abs/1710.11583* (2017).
- [13] FEAMSTER, N., REXFORD, J., AND ZEGURA, E. The road to sdn: An intellectual history of programmable networks. *Queue* 11, 12 (dec 2013), 20–40.
- [14] FERRIOL-GALMÉS, M., RUSEK, K., SUÁREZ-VARELA, J., XIAO, S., SHI, X., CHENG, X., WU, B., BARLET-ROS, P., AND CABELLOS-APARICIO, A. Routenet-erlang: A graph neural network for network performance evaluation. In *IEEE INFOCOM 2022* (2022), pp. 2018–2027.
- [15] GILMER, J., SCHOENHOLZ, S. S., RILEY, P. F., VINYALS, O., AND DAHL, G. E. Neural message passing for quantum chemistry, 2017.
- [16] GUPTA, A., MACDAVID, R., BIRKNER, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND VANBEVER, L. An Industrial-scale Software Defined Internet Exchange Point. In *Proc. USENIX NSDI* (2016).
- [17] HONG, C.-Y., MANDAL, S., AL-FARES, M., ZHU, M., ALIM, R., B., K. N., BHAGAT, C., JAIN, S., KAIMAL, J., LIANG, S., MENDELEV, K., PADGETT, S., RABE, F., RAY, S., TEWARI, M., TIERNEY, M., ZAHN, M., ZOLLA, J., ONG, J., AND VAHDAT, A. B4 and after: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN. In *Proc. ACM SIGCOMM* (2018).
- [18] JOSE, M., LAZRI, K., FRANÇOIS, J., AND FESTOR, O. Netrec: Network-wide in-network real-value computation. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)* (2022), pp. 189–197.
- [19] LAMB, I. P., DUARTE, P. A. P. R., LUIZELLI, M. C., GASPARY, L. P., AZAMBUJA, J. R., AND DA COSTA CORDEIRO, W. L. Multi-tenant programmable switch virtualization leveraging explicit resource sharing. In *2024 20th International Conference on Network and Service Management (CNSM)* (2024), pp. 1–9.
- [20] LUIZELLI, M. C., CANOFRE, R., LORENZON, A. F., ROSSI, F. D., CORDEIRO, W., AND CAICEDO, O. M. In-network neural networks: Challenges and opportunities for innovation. *IEEE Network* 35, 6 (2021), 68–74.
- [21] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., ET AL. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* 38 (2008), 67–79.
- [22] NS3. A discrete-event network simulator for internet systems, 2022. Retrieved August 20, 2022 from <https://www.nsnam.org/>.
- [23] OKTIAN, Y. E., LEE, S., LEE, H., AND LAM, J. Distributed sdn controller system: A survey on design choice. *computer networks* 121 (2017), 100–111.
- [24] PUJOL-PERICH, D., SUÁREZ-VARELA, J., FERRIOL, M., XIAO, S., WU, B., CABELLOS-APARICIO, A., AND BARLET-ROS, P. Ignition: Bridging the gap between graph neural networks and networking systems. *IEEE Network* 35, 6 (2021), 171–177.
- [25] PUJOL-PERICH, D., SUÁREZ-VARELA, J., CABELLOS-APARICIO, A., AND BARLET-ROS, P. Unveiling the potential of graph neural networks for robust intrusion detection, 2021.
- [26] RUSEK, K., SUAREZ-VARELA, J., ALMASAN, P., BARLET-ROS, P., AND CABELLOS-APARICIO, A. RouteNet: Leveraging graph neural networks for network modeling and optimization in SDN. *IEEE Journal on Selected Areas in Communications* 38, 10 (oct 2020), 2260–2270.
- [27] SAQUETTI, M., BUENO, G., CORDEIRO, W., AND AZAMBUJA, J. R. P4vbox: Enabling p4-based switch virtualization. *IEEE Communications Letters* 24, 1 (2020), 146–149.
- [28] SAQUETTI, M., CANOFRE, R., LORENZON, A. F., ROSSI, F. D., AZAMBUJA, J. R., CORDEIRO, W., AND LUIZELLI, M. C. Toward in-network intelligence: Running distributed artificial neural networks in the data plane. *IEEE Communications Letters* 25, 11 (2021), 3551–3555.
- [29] SCARSELLI, F., GORI, M., TSOI, A. C., HAGENBUCHNER, M., AND MONFARDINI, G. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80.
- [30] SCHLINKER, B., KIM, H., CUI, T., KATZ-BASSETT, E., MADHYASTHA, H. V., CUNHA, I., QUINN, J., HASAN, S., LAPUKHOV, P., AND ZENG, H. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proc. ACM SIGCOMM* (2017).
- [31] TOKUSASHI, Y., MATSUTANI, H., AND ZILBERMAN, N. Lake: The power of in-network computing. *International Conference on ReConfigurable Computing and FPGAs (ReConFig)* (2018).
- [32] WINSTEIN, K., AND BALAKRISHNAN, H. TCP Ex Machina: Computer-generated Congestion Control. In *Proc. ACM SIGCOMM* (2013).
- [33] WOODRUFF, J. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking* 28 (2020).
- [34] WU, Z., PAN, S., CHEN, F., LONG, G., ZHANG, C., AND YU, P. S. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24.
- [35] XAVIER, B. M., GUIMARÃES, R. S., COMARELA, G., AND MARTINELLO, M. Programmable switches for in-networking classification. In *IEEE INFOCOM 2021* (2021), pp. 1–10.
- [36] XIONG, Z., AND 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, 2019), HotNets '19, Association for Computing Machinery, p. 25–33.