Building a Graph-based Deep Learning network model from captured traffic traces

1st Carlos Güemes Palau
Barcelona Neural Networking Center
Universitat Politècnica de Catalunya
Barcelona, Spain
carlos.guemes@upc.edu

2nd Miquel Ferriol Galmés
Barcelona Neural Networking Center
Universitat Politècnica de Catalunya
Barcelona, Spain
miquel.ferriol@upc.edu

3rd Albert Cabellos Aparicio
Barcelona Neural Networking Center
Universitat Politècnica de Catalunya
Barcelona, Spain
albert.cabellos@upc.edu

4th Pere Barlet Ros
Barcelona Neural Networking Center
Universitat Politècnica de Catalunya
Barcelona, Spain
pere.barlet@upc.edu

***Abstract*—Currently the state of the art network models are based or depend on Discrete Event Simulation (DES). While DES is highly accurate, it is also computationally costly and cumbersome to parallelize, making it unpractical to simulate high performance networks. Additionally, simulated scenarios fail to capture all of the complexities present in real network scenarios. While there exists network models based on Machine Learning (ML) techniques to minimize these issues, these models are also trained with simulated data and hence vulnerable to the same pitfalls. Consequently, the Graph Neural Networking Challenge 2023 introduces a dataset of captured traffic traces that can be used to build a ML-based network model without these limitations. In this paper we propose a Graph Neural Network (GNN)-based solution specifically designed to better capture the complexities of real network scenarios. This is done through a novel encoding method to capture information from the sequence of captured packets, and an improved message passing algorithm to better represent the dependencies present in physical networks. We show that the proposed solution it is able to learn and generalize to unseen captured network scenarios.**

***Index Terms*—network modelling, neural networks, graph neural networks, recurrent neural networks**

I. Introduction

In recent years, network modeling has risen in prominence as one of the most active research fields related to computer networks. Correctly designed network models can be used to simulate network configurations without risk, as it does not involve using the actual network, and try out scenarios that may be too rare or risky to encounter in real life. Arguably, the most prevalent way to build these network models is through the use of discrete event simulation (DES) methodologies. Notable examples include the ns-3 [1] and OMNeT++ [2] simulators. However, while DES-based simulators tend to be accurate, they also tend to be slow and scale poorly with the simulation scenario size (e.g., amount of traffic, size of the network...) [3]. Also, simulation tends to be bounded by only considering idealized scenarios in which all variables and patterns are known, which usually it is not the case in real life (e.g. the traffic distribution and its parameters [4]–[6]).

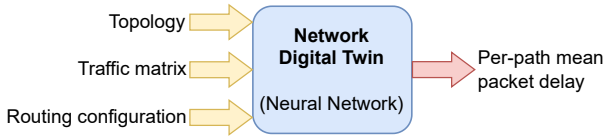


Fig. 1: Scheme of the neural network-based solution requested in the Graph Neural Networking Challenge

Hence, as of late, there has been a push to complement DES with Machine Learning (ML) methodologies. These hybrid techniques offer similar accuracy while offering a speed up in performance by replacing part of the simulation with ML models [7], [8]. Alternatively, there are end-to-end ML models that entirely replace DES simulators [9], [10]. These tend to be Graph Neural Networks (GNNs) [11], as it is a type of neural network (NN) that is able to exploit relational information and therefore able to understand the inter-dependencies between the different devices in the network, independently of its topology and size. Still, the ML models present in these solutions are trained with DES-generated data, resulting in them inheriting the same constraints and challenges associated with DES simulators.

The objective behind the 2023 edition of the Graph Neural Network Challenge is the creation of a network model that does not share the same limitations as DES simulators. This means the creation of a network model purely from captured data, rather than using DES-simulators or DES-generated data. To do so, we have built a testbed server that is able to simulate different network topologies under different levels of load and two traffic flow distributions. The result is a

This publication is part of the Spanish I+D+i project TRAINER-A (ref.PID2020-118011GB-C21), funded by MCIN/AEI/10.13039/501100011033. This work is also partially funded by the Catalan Institution for Research and Advanced Studies (ICREA), and by the Joan Oró predoctoral program, from the Secretariat for Universities and Research, part of the Ministry of Research and Universities of the Goverment of Catalonia, and the European Social Fund Plus (ref. BDNS 657443).

dataset formed from 8604 samples, each sample representing one traffic scenario in the testbed. Participants of the challenge are asked to design GNN-based models using this dataset, as summarized in Figure 1.

In this paper, we present one possible solution for the Graph Neural Network Challenge 2023. Our proposal is the continuation of the RouteNet family of models, in which we introduce two novel adaptations for dealing with captured data. On one hand, we introduce a new way of encoding the network traffic distribution through the use of a Recurrent Neural Network (RNN), a type of neural network meant to deal with sequences of data. On the other, we also updated the GNNs message passing paradigm to be better suited for the more complex real networks. The results show that with these changes, our model achieves significant improvements in modeling packet delays of captured traffic data. The remainder of the paper is structured as follows:

- In section II we briefly introduce GNNs and RNNs and their notations.
- In section III, we introduce the proposed solution to the Graph Neural Network Challenge 2023.
- In section IV, we evaluate the proposed solution relative to a baseline model based on the previous generation of the RouteNet models, RouteNet-Fermi [10].

II. Background

A. Graph Neural Networks

Relational information can be expressed in the form of a graph $G \in \{V, E\}$, itself identified by a set of vertices $v \in V$ connected by a set of edges $e \in E$. Hence, GNNs [11] are the family of NNs designed precisely to work with and exploit relational information. They have been successfully applied in many fields in which relational information is available, which also includes network modeling [12], [13].

There are plenty of GNN variants, each defined with a different mechanism to extract and utilize relational information [12]. For example, spectral GNNs update the encoding for each vertex through a convolution operation based on the spectrum of the Laplacian matrix [14], while spatial GNNs define its convolution using the vertex's neighborhood in the original topology [15]. Independently of the specific GNN architecture used, the fact that its mechanism is equivariant to vertex and edge permutations grants GNNs *strong relational inductive bias*, that is, the ability to generalize across topologies [16].

Specifically, our proposed solution is based on Message-Passing NN (MPNN) [17]. They are defined as a more general framework of spatial GNNs which identify their models in three phases: an initial encoding phase where the initial representation for each vertex is obtained, followed by a message passing phase in which neighboring vertices share information, and ending in the readout phase in which the vertices' states are used to generate the final output. Specifically, the message-passing phase is defined as the following steps:

1) Message generation: each node $v \in V$ generates a message to its neighbors $w \in N_v, N_v \subseteq V$ using both nodes' states h_v and h_w, and edge information $e_{(v,w)}$:
$$m_{w \to v} = M(h_v, h_w, e_{(v,w)}) \quad \text{iff } w \in N_v$$
2) Message aggregation: each node aggregates the messages from its neighbors:
$$m_v = agg(\{m_{w \to v}, w \in N_v\})$$
3) Update: the aggregated messages are used to update the current node state:
$$h_v = U(h_v, m_v)$$

This process is repeated for a fixed number of iterations. Both functions M and U are learnable, differentiable functions that can be implemented by NNs. The aggregation operator can be any commutative operation, such as the sum of all messages, as (generally) neighboring vertices are not described by an order relation.

B. Recurrent Neural Networks

RNNs [18] are a type of state-full NNs. Unlike most NNs, which can be represented as a pure mathematical function $y = f(x)$, whose output y is only affected by its input x, RNNs also utilize a state h. This makes RNNs very useful for dealing with data structured as a sequence, such as time series. Also, unlike other types of models (e.g. transformers), it can also accept input sequences of undetermined length, which makes them highly versatile. To use an RNN, besides the input x the RNN also requires the current state h. The RNN will not only produce the output y, but also a modified state h':

$$y, h' = RNN(x; h)$$

In order to process a sequence $X = [x_0, ..., x_n]$, it can be thought of as the RNN iterates through every single element to return the output sequence Y:

$$Y, h' = RNN(X; h) \Leftrightarrow \begin{array}{l} y_0, h' = RNN(x_0; h) \\ \textbf{for } i \in [1, n] \textbf{ do:} \\ \quad y_i, h' = RNN(x_i; h') \end{array}$$

Nowadays the most commonly used architectures for RNN are the Long Short-Term Memory (LSTM) [19] and the more recent Gated Recurrent Unit (GRU) [20] architectures, which offers similar expressive power as the former with fewer parameters [21]. In our solution, we use RNNs in instances where the information is structured as sequences, such as when encoding packet traffic or when processing the evolution of a traffic flow as it crosses through the network.

III. Our proposed solution

In this section, we introduce our proposed solution for the Challenge, a GNN-based solution designed to handle the difficulties of real network data. To do so, the model includes two improvements over its predecessor. First, the message-passing algorithm has been expanded to capture the complexities of real network scenarios. Second, our proposed

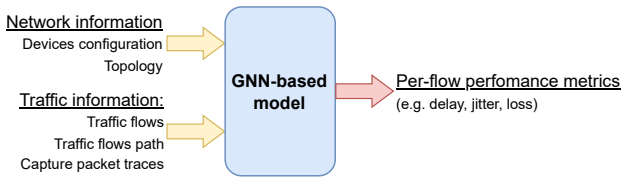


Fig. 2: Black-box scheme of the proposed GNN model

solution also includes using an RNN-based encoder that can process complex, non-parameterized traffic distributions through sequences of captured packets.

Figure 2 shows a black-box representation of this model. The model takes as input the characteristics of the network scenario. These characteristics can be divided into two groups:

- Characteristics of the underlying network: its devices, including links, routers, and switches; and its topology.
- Characteristics of the traffic flows: their source, their destination, their path across the network, and their size both in terms of the number of bits transmitted and the number of packets. We also include sequences of captured packets, that will later be used to learn about the underlying traffic distributions.

The output of the model focuses on obtaining per-flow performance metrics, although in the context of the challenge and this paper, we will focus on obtaining the per-flow mean delay.

Ultimately, the objective of the model is to exploit an accurate representation of the network components to generalize to unseen topologies and scenarios, while capturing the complexities of captured traffic data. In this section, we will begin by explaining how we formally represent network components and their relationships. We will then follow by explaining how the initial representations for the network elements are obtained. At the end of the section, we will summarize our proposed solution.

A. Representing network components and dynamics

First, let us define a physical network as a set of devices $\mathcal{D} \in \{d_i : i \in (1, ..., n_d)\}$ connected through a set of links $\mathcal{L} = \{l_i : i \in (1, ..., n_l)\}$. Each device d_i is composed by a set of ports $\mathcal{Q} = \{q_i : i \in (1, ..., n_q)\}$, such as $d_i \subset \mathcal{Q}$. Physical links and ports have a 1-to-1 relationship, that is, each physical link is connected to a unique port, and each port is connected to a unique link. In the context of the challenge's networks, these devices can either be switches or routers.

A network is also populated by a set of flows $\mathcal{F} = \{f_i : i \in (1, ..., n_f)\}$. Each flow is defined by the path of links and ports that it traverses through the network, forming a sequence of tuples of ports and links, defined as $f_i = \{(q_{p_q(f_i,0)}, l_{p_l(f_i,0)}), ..., (q_{p_q(f_i,|f_i|)}, l_{p_l(f_i,|f_i|)})\}$, where p_q and p_l return the index of the j-th port or link along the path of flow f_i, respectively. For commodity later on, we will also define the function $Q_f(q_i)$, which returns all the flows that go through port q_i and its associated link, and the function $Q_d(q_i)$ which returns the device which the port q_i belongs to.

From the network elements we have defined, we can build a graph that represents the relationships between links, ports, devices, and flows. However, we must also represent the following network dynamics:

(i) Each link is associated with a single port, each port is associated to a single link.
(ii) The state of ports of the same device will be affected by each other due to their dependencies on shared resources (e.g., a device's shared memory and computing power).
(iii) The state of flows (e.g. throughput, losses) is affected by the state of the links and ports they traverse.
(iv) The state of links and ports (e.g. link utilization, port queue utilization) will depend on the state of flows passing through them.

Now, we present how these properties can be formalized:

(i) Each link is associated to a single port, each port is associated to a single link: the most straightforward way to represent this property is to encode each port-link pair together with h_{lq_i}, where h_{lq_i} is the encoded state for the port-link pair $lq_i \equiv (l_i, q_i)$. While another solution would have been to define a set of functions that maps between the two, this would have increased the complexity of our proposed solution without necessarily increasing its expressive power.

(ii) The state of ports of the same device will be affected by each other due to their dependencies on shared resources:

$$h_{d_i} = G_d(\{h_{lq_j}, q_j \in d_i\}) \tag{1}$$

Where h_{d_i} is the encoded state of the device d_i, and G_d is an unknown function that takes as input all the states belonging to the ports of d_i and outputs its encoded state.

(iii) The state of flows is affected by the state of the links and ports they traverse:

$$h_{f_i} = G_f([h_{Q_d(q_{f_i(1)})}, h_{lq_{f_i(1)}}, ..., h_{Q_d(q_{f_i(|f_i|)})}, h_{lq_{f_i(|f_i|)}}]) \tag{2}$$

Where h_{f_i} is the encoded state of the flow f_i, and G_f is an unknown function that takes as input the sequence of states belonging to the devices, links, and ports that flow f_i traverses and outputs its encoded state. Specifically, for the j-th position in f_i's path, $h_{Q_d(q_{f_i(j)})}$ represents the state of the device which the j-port belongs to, and $h_{lq_{f_i(j)}}$ represents the state of the j-link-port pair.

(iv) The state of links and ports will depend on the state of flows passing through them:

$$h_{lq_i} = G_{lq}(\{h_{f_j}, f_j \in Q_f(q_i)\}) \tag{3}$$

Where G_{lq} is an unknown function that takes as input all the encoded states from flows that go through the link-port lq_i and returns the encoded state for it.

Note that these properties show circular dependencies between the different states of the elements in the network. As a result, we will use an iterative algorithm that starts with the initial set of encoded states h^0_{lq}, h^0_f, h^0_d, and in each iteration they are refined until convergence. This will be the message-passing algorithm, although its description will be introduced later in subsection III-C.

B. Extracting initial embedding

However, before we can run the iterative algorithm to refine the encoded states, we need to obtain the initial embeddings for each of the network elements. These embeddings are defined using the elements' characteristics within the network scenarios.

Network Device	Features	Notes
Link information	Bandwidth	In bits per second
Device information	Device type	Router or Switch
Flow	Average load	In bits per second
	Number of packets	
	Packet size	In bits
	Sequence of packets	Features per packet: timestamp and packet size

TABLE I: Extracted features of each network element for initial embeddings

The extracted features per network device can be seen in Table I. Note that there may be other relevant features that we have not included. For example, features like the flow's quality of service or the link's propagation delay have not been included as they are constant across the entire dataset, but may be included if needed. Other features, such as each port's buffer memory size, have not been included since it is not information that is usually available to the network operator, and therefore unrealistic to be added to the network model.

Out of the identified features, the link bandwidth, and the flow's average load, packet size and number of packets have all been normalized with min-max normalization. The device type is a categorical feature with two possible values. For our solution have decided to separate ports and device encoding functions into two separate functions, one for each type of device. This grants us more expressive power than encoding the device type through one-hot encoding. While it scales poorly to the number of devices, this is not an issue as in Challenge's dataset only considers two types of devices.

To encode the link and port features, the link and device features are used with the encoding function E_{lq}, to obtain the initial embeddings:

$$h^0_{lq_i} = E_{lq}(\{\text{Link bandwidth}, \text{Port's device type}\})$$

$$= \begin{cases} E_{lq,router}(\{\text{Link bandwidth}\}) & \text{if } q_i \text{ belongs to router} \\ E_{lq,switch}(\{\text{Link bandwidth}\}) & \text{if } q_i \text{ belongs to switch} \end{cases} \tag{4}$$

Then, to obtain the device embeddings, all the port embedding belongings to the same device are aggregated through the sum operator followed by the encoding function E_d:

$$h^0_{d_i} = E_d(sum(\{h_{lq_j}, q_j \in d_i\}))$$

$$= \begin{cases} E_{d,router}(sum(\{h_{lq_j}, q_j \in d_i\})) & \text{if } d_i \text{ is router} \\ E_{d,switch}(sum(\{h_{lq_j}, q_j \in d_i\})) & \text{if } d_i \text{ is switch} \end{cases} \tag{5}$$

To encode the traffic flows, we first need to encode their sequences of packets. To do so, we apply the following process for each flow f_i:

1) First we trim the sequence to only include the first second of captured traffic. We then aggregate the sequence, where we obtain the number packets sent per millisecond. As in the Challenge's dataset the packet size in each flow is constant, this is equivalent to the number of bits transmitted. This reduces the length of the sequence significantly, which makes it faster and easier to handle by the model later on.
2) We train a sequential encoder model E_{pkts} to process the entire sequence into a single encoded vector h_{pkts_i}.

$$h_{pkts_i} = E_{pkts}(agg(\{\text{Sequence of packets}\}))$$

3) We input the encoded vector with the average load, number of packets, and average packet size into an encoder function E_f to obtain the initial flow embeddings.

$$h^0_{f_i} = E_f(h_{pkts_i} | \{\text{Avg. load, Avg. packet size, Num. packets}\}) \tag{6}$$

C. Summary of the model

The summary of the solution is described at Algorithm 1. To do so, our model can be divided into three phases, following the MPNN paradigm:

1) Initial encoding (lines 1-15): initial embeddings are built from the network and traffic flows's features.
2) Message passing (lines 17-34): the different embeddings are enriched through our multi-stage message passing algorithm which represents the network's dynamics.
3) Readout (lines 36-40): the final traffic embeddings are used in this phase to obtain the final packet delay.

The initial encoding phase is implemented as described back in subsection III-B. Specifically, lines 3-7 are dedicated to obtaining the flow embeddings, lines 9-11 the link and port embeddings, and lines 13-15 the device embeddings. The encoding functions for flows E_f, links and ports E_{lq}, and devices E_d, are a single Multilayer Perceptron (MLP) each. The sequential model used to encode the packet sequences (lines 4-6) is a two-layered RNN (i.e., equivalent to two RNNs where the output sequence of the first is used as the input sequence of the second one) using GRU cells. To do so, we introduce the entire pre-processed packet sequence, and we retrieve the final internal state from the second layer.

After building the initial representations the model begins the message-passing phase. This is an iterative process and exploits the properties described back in subsection III-A. At each iteration, the embeddings are updated as follows:

1) The flow embedding (lines 20-26): Here we use a GRU-cell RNN model RNN_{flows} to learn function G_f as described back in Equation (2). RNN_{flows} takes the sequence of internal states from links, ports, and devices, and returns the sequences of partial flow state after each point in the sequence $\widetilde{m}^t_f$. The final internal state of the RNN acts as the new flow embedding h^t_f. As such, the entire process with RNN_{flows} includes the aggregation, update, and message generation steps of the message-passing paradigm.

Algorithm 1 Summary of our solution

Input: $\mathcal{F}, \mathcal{Q}, \mathcal{L}, \mathcal{D}, \boldsymbol{x}_f, \boldsymbol{x}_{f,pkts}, \boldsymbol{x}_q, \boldsymbol{x}_l$
Output: $\hat{\boldsymbol{y}}_f$

1: ▷ Initial encoding phase ◁
2: **for all** $f \in \mathcal{F}$ **do** ▷ Initial Flow Embeddings
3: $\boldsymbol{h}_{f,pkts} = \mathbf{0}_{pkts}$ ▷ Encoder RNN initial state is a zero vector
4: **for all** $x_{f,pkt} \in \boldsymbol{x}_{f,pkts}$ **do**
5: $\emptyset, \boldsymbol{h}_{f,pkts} = RNN_{pkts}(x_{f,pkt}; h_{f,pkts})$
6: $\boldsymbol{h}_f^0 \leftarrow E_f(\boldsymbol{x_f}|\boldsymbol{h}_{f,pkts})$
7: $\mathcal{LQ} \equiv \{(l,q), l \in L \wedge q \in Q \wedge paired(l,q)\}$ ▷ Set of paired links and ports
8: **for all** $(l,q) \in \mathcal{LQ}$ **do** $\boldsymbol{h}_{lq}^0 \leftarrow E_{lq}(\boldsymbol{x_l}|\boldsymbol{x_q})$ ▷ Initial Link and Port Embeddings
9: **for all** $d \in D$ **do** ▷ Initial Device Embeddings
10: $M_d^0 \leftarrow \sum_{(l,q)\in\mathcal{LQ}\wedge q\in d} \boldsymbol{h}_{lq}^0$
11: $\boldsymbol{h}_d^0 \leftarrow E_d(M_d^0)$
12: ▷ Message Passing Phase ◁
13: $t = 0$
14: **repeat**
15: $t \leftarrow t + 1$
16: **for all** $f \in \mathcal{F}$ **do** ▷ Message Passing on Flows
17: $pos \leftarrow 0$ ▷ Curr. position in path
18: $\boldsymbol{h}_f^t \leftarrow \boldsymbol{h}_f^{(t-1)}$
19: **for all** $(l,q) \in f$ **do**
20: $\widetilde{m}_{f,pos}^t, \boldsymbol{h}_f^t \leftarrow RNN_{flows}(\boldsymbol{h}_{lq}^{t-1}; \boldsymbol{h}_f^t)$
21: $pos \leftarrow pos + 1$
22: $\widetilde{m}^t \leftarrow \{\widetilde{m}_f^t, f \in \mathcal{F}\}$
23: **for all** $(l,q) \in \mathcal{LQ}$ **do** ▷ MP on Links and Ports
24: $M_{lq}^t \leftarrow \sum_{(f,pos)\in\hat{Q}_f(l,q)} \widetilde{m}_{(f,pos)}^t$ ▷ L&Q aggregation
25: $\boldsymbol{h}_{lq}^t, \emptyset \leftarrow U_q(M_{lq}^t; \boldsymbol{h}_{lq}^{t-1})$ ▷ L&Q update
26: $\widetilde{m}_{lq}^t \leftarrow \boldsymbol{h}_{lq}^t$ ▷ L&Q message generation
27: **for all** $d \in \mathcal{D}$ **do** ▷ Message Passing on Devices
28: $M_d^t \leftarrow \sum_{(l,q)\in\mathcal{LQ}\wedge q\in d} \widetilde{m}_{lq}^t$ ▷ Device aggregation
29: $\boldsymbol{h}_d^t, \emptyset \leftarrow U_q(M_d^t; \boldsymbol{h}_d^{t-1})$ ▷ Device update, msg. generation
30: **until** $t > T \vee \widetilde{m}^t \simeq \widetilde{m}^{t-1}$
31: ▷ Readout Phase ◁
32: **for all** $f \in \mathcal{F}$ **do**
33: $\hat{d}_q \leftarrow \sum_{\widetilde{m}_{f,pos}^t \in \widetilde{m}_f^t} R(\widetilde{m}_{f,pos}^t)$ ▷ Queuing delay
34: $\hat{d}_t \leftarrow \sum_{(l,q)\in f} \boldsymbol{x}_{f_{ps}}/\boldsymbol{x}_{l_c}$ ▷ Transmission delay
35: $\hat{d}_p \leftarrow \sum_{(l,q)\in f} \boldsymbol{x}_{l_{prop}}$ ▷ Propagation delay
36: $\hat{\boldsymbol{y}}_{f_d} \leftarrow \hat{d}_q + \hat{d}_t + \hat{d}_p$

2) The link and port embedding (lines 27-30): These lines replicate the behavior described back in Equation (3). For each flow, the adequate partial flow states $\widetilde{m}_{f,pos}^t$ are retrieved and aggregated through addition (line 28). To correctly retrieve the partial flow states we defined function $\hat{Q}_f(l,q)$, which given a link-port pair (l,q) as input it returns the set of flows traversing through them, as well as their position in the flow's path. The update of the link and port state h_{lq}^t is done through the function U_q, implemented as a GRU cell, which takes as input the aggregated messages and as initial state the previous iteration's embedding h_{lq}^{t-1} (line 29).
3) The device embedding (lines 31-33): this step is nearly identical to updating the link and port embedding, while representing the property described in Equation (1).

The iterative process runs for a maximum of T iterations, or until the sequences of partial flow embeddings $\widetilde{m}^t$, which will be later used, have converged (line 34). We determine convergence if the mean-absolute relative difference between $\widetilde{m}^t$ and $\widetilde{m}^{t-1}$ is lower than 5% in 95% of flows in the scenario. These values are hyperparameters of the solution.

Finally, in the readout phase, the final mean packet delay $\hat{y}_d$ is computed for each flow. The delay is divided into the queuing delay, transmission delay, and propagation delay, which are later added together (line 40). The transmission delay (line 38) and propagation delay (line 39) can be computed directly from the flow features, as we know the average flow bandwidth, link capacity, and link propagation delay. For computing the queuing delay (line 37), each of the flow partial states is passed through the readout function R to obtain the marginal queuing delay attributed at each link-port pair, and then added. Function R is modeled through an MLP.

IV. Evaluation

In the following section, we cover how we evaluated our solution. Specifically, we first introduce the testbed and the characteristics of the generated dataset. Then, we compare how accurate the proposed solution is when predicting the mean flow delay compared to a baseline model, itself inspired by a naive adaptation of RouteNet-Fermi [10] before introducing the improvements discussed in Section III.

A. Testbed environment

To obtain captured traffic traces needed for the Challenge, a testbed network was built to first generate them. Figure 3 shows the structure of the testbed and its components. How the testbed works is summarized as follows:

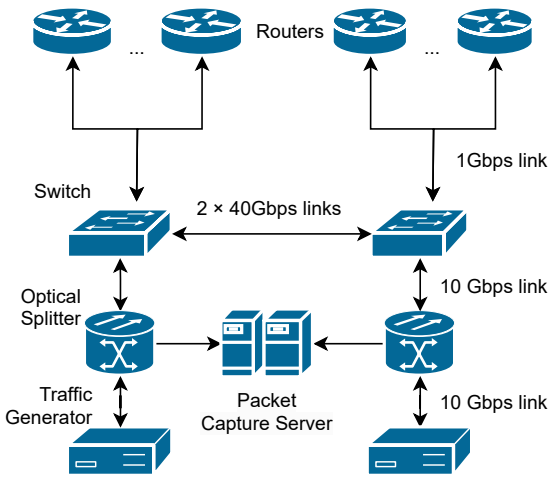


Fig. 3: Diagram summarizing the testbed's structure

- The testbed can be formed with up to 8 Huawei NetEngine 8000 M1A routers connected to one of the two Huawei S5732-H48UM 2CC 5G Bundle switches.
- The switches act as the backbone of the testbed, connecting the routers, the traffic generators and packet capture servers to each other.
- Two TRex traffic generators send traffic in such a way that it passes through the indicated routers before ending in either of the traffic generators.
- Traffic between the traffic generators and the switches is copied using an optical splitter and then fed to the packet capture server using the PF_RING software.

The traffic generators can generate two traffic distributions:

- Constant Bit Rate (CBR): short, regular, high-frequency bursts that average out to a specified flow rate.
- Multi-Burst (MB): similar to CBR, but rather than transmitting the entire flow at once, traffic is characterized by periods of no activity interrupted by fixed-length bursts (in terms of number of packets).

The configuration based on switches allows for the testbed to vary the topology of the network with ease. In each network scenario, a topology is determined and fixed, and multiple flows are generated. Scenarios can be split into two groups: scenarios that include only MB flows, and scenarios that include both CBR and MB flows. In the first group, there can only be one flow per source-destination pair (as in the first and last router in the path). In the second group, there can be up to one CBR flow and one MB flow per source-destination pair. Also, if two flows share the same source-destination pair they also share the same routing path.

Scenarios lasted for 10 seconds, but only packets sent in the last 5 seconds were captured. This is due that at the beginning of the scenario, the network remains in a transient state, where flows are just starting, buffers are still being filled, and the flow rates are still too irregular. Also, due to the cost of storing all the packet traces and simulating the scenarios, there is a limit to the amount of samples we can generate. Table II shows how many samples were used for training, validation, and testing, and how many of which were used for each scenario.

Scenario type	**Training**	**Validation**	**Test**
CBR+MB	3372	843	150
MB	3512	877	150
Total	*6884*	*1720*	*300*

TABLE II: Number of samples used for training and evaluation

B. Training and results

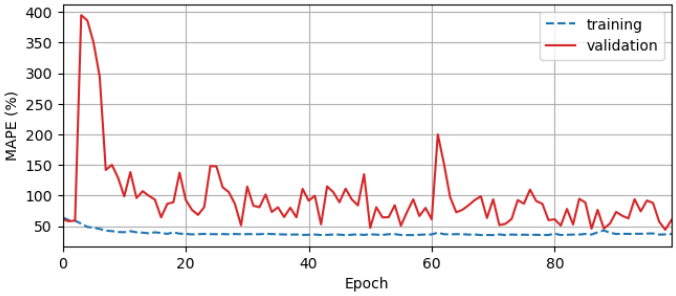
We implemented both our solutions using Tensorflow 2.11. The baseline model's architecture is based on RouteNet-Fermi's architecture [10] but with small changes to work in the testbed environment. Specifically, the message-passing component only considers links and flows, and the flow embeddings are defined using each flow's size in bits, its number of packets, the size of those packets, and its distribution (either CBR or MB) represented by two one-hot encoded features.

Hyperparamaters	Baseline	Our solution
Flow embedding size	64	
Link embedding size	64	
Device embedding size	-	16
MP max iterations	8	40
MP convergence threshold	-	0.05
Learning rate	10^-3	2.5×10^-4
Number of training epochs*	98	86
Loss function	MAPE	log MSE

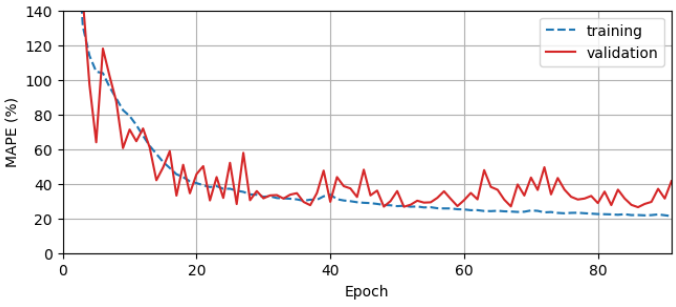
TABLE III: Hyperparameters of both models

After iterating with multiple combinations of hyperparameters and several grid searches, we decided on the hyperparameter values present in table III. In the case of the number of epochs, both models were allowed to run with a redundant amount of epochs, and the best model was taken from the epoch in which the validation error was minimized (the table reflects at which epoch this occurred). Note the baseline model does not consider node embeddings, and only ends the message-passing process after a fixed number of iterations.

Another difference between both models is the loss function used: the original model uses the Mean Absolute Percentage Error (MAPE), as it is the metric we wish to minimize. The MAPE has the advantage of being a relative metric, making it independent of the scale of the true values. However, using it as the loss function introduces a bias where the model tends to underestimate its predictions [22]. As a result, our proposed solution is instead trained using the MSE of the logarithm of both the prediction and true value.



(a) Baseline model



(b) Our proposed solution

Fig. 4: Loss evolution during training

Model	Best epoch	Validation MAPE	Test MAPE
Baseline	98	44.298%	*100.528%*
Proposed solution	86	26.449%	*27.831%*

TABLE IV: Test partition MAPE scores of both the baseline model and proposed solution

The results of training both the baseline and our proposed solution can be examined in Figure 4. Specifically, Figure 4a shows the training and validation MAPE as training evolves in the baseline model, while Figure 4b shows the evolution of the MAPE in the proposed solution. Overall, it is clear that the baseline model is inadequate to learn the complexities of the dataset only reaching a validation MAPE of 44%, while our proposed solution does show the ability to learn and generalize, reaching a lower bound of 26.449% MAPE. These results are further confirmed in Table IV, which shows how the best epoch from each model fares against the test dataset. While the baseline fails to generalize, and scores a MAPE barely over 100% in the test dataset, the proposed solution correctly generalizes and gets a MAPE of 27.831%.

V. Related Work

Current state-of-the-art network models are varied in how they balance accuracy and computational cost. Overall, we might find that network models fall into one of three approaches: purely simulator-based, purely Machine Learning (ML) based models, and simulator-ML hybrids.

Network models based on simulators tend to be the most common, due to their high accuracy albeit at the cost of high simulation times, which increases exponentially to the scenario size. This includes popular simulators like ns-3 [1] and OMNeT++ [2], both based on Discrete Event Simulation (DES). New simulators have been published, such as DONS [23], with improved implementations that decrease computation costs and allow for the simulation to be parallelized, but are still limited by the poor scaling to simulation size. On the other hand, trace-based simulation like Parsimon [24] is faster than DES, but at the cost of lower accuracy.

Next, simulator-ML hybrid network models aim to replace part of the simulation with ML models. Doing so decreases simulation time, by avoiding to simulate segments of the network, while also opening the possibility of exploiting parallelization, another a major drawback of DES. However this comes at the cost of having a marginally worse accuracy. This hybrid approach was first popularized by MimicNet [7] and it was later expanded and improved on by DeepQueueNet [8]. While faster than pure simulation, these approaches still struggle to simulate larger network scenarios at a reasonable computational cost. Additionally, just as pure simulators, these solutions only consider idealized scenarios that may not reflect real network scenarios encountered by network administrators.

Finally, there exist pure ML network models, which are trained using simulators that generate its samples, but then do not require simulation during inference. This last property makes ML models' inference several degrees of magnitude faster than both simulation and hybrid models. While multiple NN architectures were tested [25], [26], eventually GNNs came out as the most appropriate architecture due to their ability to generalize across unseen network topologies. An example of these is the RouteNet family of models [9], [27], with RouteNet-Fermi being its latest iteration [10]. These models do not only infer quickly but are also adapted to deal with networks much larger than the ones seen during training. However, this comes at the cost of being less accurate than the other approaches. Also, as they are trained using simulated data, they also learn to only consider idealized scenarios and may expect certain information as input which in practice would be unknown to the network operators.

VI. Conclusion

In this paper, we have presented a new GNN-based architecture for network modelling designed to be trained with captured traces rather than with simulated ones, as a possible solution to the Graph Neural Network Challenge 2023. In it, we have seen how we can encode packet traffic to better represent traffic distributions, and a new message passing algorithm that is designed to better model the relationships between traffic flows, physical links, devices and their ports.

Yet, there are aspects of the proposed solution that we wish to study further. For example, while we propose a method to encode packet sequences, many of the design decisions must be further studied. This includes how the sequences were summarized by aggregating information by milliseconds, or the use of a two-layered RNN to encode its information rather than any other sequential model like a bidirectional RNN or a transformer. Similarly, while in the Challenge we do not provide the packet traffic during the start of the simulation, as in theory this traffic represents an unstable and transient state of the network, it would be interesting to study a way of extracting useful information from it as well. Additionally, this Challenge we have simplified certain aspects of the problem, such as including scenarios with queuing policies, and will eventually need to be revisited. Ultimately, we have decided to leave all of these questions open for future work.

References

[1] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*, pp. 15–34. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[2] A. Varga, *A Practical Introduction to the OMNeT++ Simulation Framework*, pp. 3–51. Cham: Springer International Publishing, 2019.

[3] S. Jafer, Q. Liu, and G. Wainer, "Synchronization methods in parallel and distributed discrete-event simulation," *Simulation Modelling Practice and Theory*, vol. 30, pp. 54–73, 2013.

[4] E. Kresch and S. Kulkarni, "A poisson based bursty model of internet traffic," in *2011 IEEE 11th International Conference on Computer and Information Technology*, pp. 255–260, 2011.

[5] V. Paxson and S. Floyd, "Wide area traffic: the failure of poisson modeling," *IEEE/ACM Transactions on Networking*, vol. 3, no. 3, pp. 226–244, 1995.

[6] J. Popoola and R. A. Ipinyomi, "Empirical performance of weibull self-similar tele-traffic model," *International Journal of Engineering and Applied Sciences*, vol. 4, 2017.

[7] Q. Zhang, K. K. W. Ng, C. Kazer, S. Yan, J. a. Sedoc, and V. Liu, "Mimicnet: Fast performance estimates for data center networks with machine learning," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, (New York, NY, USA), p. 287–304, Association for Computing Machinery, 2021.

[8] Q. Yang, X. Peng, L. Chen, L. Liu, J. Zhang, H. Xu, B. Li, and G. Zhang, "Deepqueuenet: Towards scalable and generalized network performance estimation with packet-level visibility," in *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, (New York, NY, USA), p. 441–457, Association for Computing Machinery, 2022.

[9] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "Routenet: Leveraging graph neural networks for network modeling and optimization in sdn," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2260–2270, 2020.

[10] M. Ferriol-Galmés, J. Paillisse, J. Suárez-Varela, K. Rusek, S. Xiao, X. Shi, X. Cheng, P. Barlet-Ros, and A. Cabellos-Aparicio, "Routenet-fermi: Network modeling with graph neural networks," 2022.

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

[12] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Comput. Surv.*, vol. 54, oct 2021.

[13] J. Suárez-Varela, P. Almasan, M. Ferriol-Galmés, K. Rusek, F. Geyer, X. Cheng, X. Shi, S. Xiao, F. Scarselli, A. Cabellos-Aparicio, and P. Barlet-Ros, "Graph neural networks for communication networks: Context, use cases and opportunities," *IEEE Network*, vol. 37, no. 3, pp. 146–153, 2023.

[14] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," 2014.

[15] A. Micheli, "Neural network for graphs: A contextual constructive approach," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 498–511, 2009.

[16] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018.

[17] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, p. 1263–1272, JMLR.org, 2017.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, 1986.

[19] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.

[20] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014.

[21] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014.

[22] J. McKenzie, "Mean absolute percentage error and bias in economic forecasting," *Economics Letters*, vol. 113, no. 3, pp. 259–262, 2011.

[23] K. Gao, L. Chen, D. Li, V. Liu, X. Wang, R. Zhang, and L. Lu, "Dons: Fast and affordable discrete event network simulation with automatic parallelization," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, (New York, NY, USA), p. 167–181, Association for Computing Machinery, 2023.

[24] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson, "Scalable tail latency estimation for data center networks," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), pp. 685–702, USENIX Association, Apr. 2023.

[25] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, (New York, NY, USA), p. 185–191, Association for Computing Machinery, 2017.

[26] S. Xiao, D. He, and Z. Gong, "Deep-q: Traffic-driven qos inference using deep generative network," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, NetAI'18, (New York, NY, USA), p. 67–73, Association for Computing Machinery, 2018.

[27] M. Ferriol-Galmés, K. Rusek, J. Suárez-Varela, S. Xiao, X. Shi, X. Cheng, B. Wu, P. Barlet-Ros, and A. Cabellos-Aparicio, "Routenet-erlang: A graph neural network for network performance evaluation," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pp. 2018–2027, 2022.