Computer Networks 217 (2022) 109329

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Building a Digital Twin for network optimization using Graph Neural Networks☆

Miquel Ferriol-Galmés [a,*], José Suárez-Varela [a], Jordi Paillissé [a], Xiang Shi [b], Shihan Xiao [b], Xiangle Cheng [b], Pere Barlet-Ros [a], Albert Cabellos-Aparicio [a]

[a] Barcelona Neural Networking Center, Universitat Politècnica de Catalunya, Spain
[b] Huawei Technologies Co., Ltd., China

ARTICLE INFO

Keywords:
Digital Twin
Graph Neural Networks
Network optimization
Deep Learning
Network modeling

ABSTRACT

Network modeling is a critical component of Quality of Service (QoS) optimization. Current networks implement Service Level Agreements (SLA) by careful configuration of both routing and queue scheduling policies. However, existing modeling techniques are not able to produce accurate estimates of relevant SLA metrics, such as delay or jitter, in networks with complex QoS-aware queueing policies (e.g., strict priority, Weighted Fair Queueing, Deficit Round Robin). Recently, Graph Neural Networks (GNNs) have become a powerful tool to model networks since they are specifically designed to work with graph-structured data. In this paper, we propose a GNN-based network model able to understand the complex relationship between (*i*) the queueing policy (scheduling algorithm and queue sizes), (*ii*) the network topology, (*iii*) the routing configuration, and (*iv*) the input traffic matrix. We call our model TwinNet, a *Digital Twin* that can accurately estimate relevant SLA metrics for network optimization. TwinNet can generalize to its input parameters, operating successfully in topologies, routing, and queueing configurations never seen during training. We evaluate TwinNet over a wide variety of scenarios with synthetic traffic and validate it with real traffic traces. Our results show that TwinNet can provide accurate estimates of end-to-end path delays in 106 unseen real-world topologies, under different queuing configurations with a Mean Absolute Percentage Error (MAPE) of 3.8%, as well as a MAPE of 6.3% error when evaluated with a real testbed. We also showcase the potential of the proposed model for SLA-driven network optimization and what-if analysis.

1. Introduction

Network optimization is typically achieved by combining two main elements: (*i*) a network model and (*ii*) an optimization algorithm (e.g., [1]). The model predicts the performance (e.g., per-path delay) for a specific configuration (e.g., routing), while the optimization algorithm generates configurations that can potentially meet the expected performance, for example, according to a Service Level Agreement (SLA).

Emerging use cases have renewed interest in SLA optimization [2]. SD-WAN [3], 5G network slicing [4], networked control of industrial systems, such as Industry 4.0 [5] and Tactile Internet [6,7], require new stringent SLA requirements. In addition, novel forms of communication, such as AR/VR or holographic telepresence, demand ultra-low deterministic latency [8,9]. In order to efficiently offer such SLAs, network optimizers must consider both routing *and* queue scheduling mechanisms (e.g., Strict Priority, Weighted Fair Queueing, Deficit Round Robin).

A fundamental aspect of network optimization is that *we can only optimize what we can model.* For example, to optimize the delay of a path traversing some nodes with different queuing policies, the model must be able to understand how delay relates to queuing policies and traffic.

Network modeling is a well-established topic and, as such, we have witnessed a rich body of proposals in the literature [10,11]. The most well-known models are analytical models, fluid models and packet-level simulators. First, analytical models based on queuing theory have been extensively used for this purpose [12]. However, queuing theory struggles to model accurately realistic networks with multi-hop routing,

☆ This publication is part of the Spanish I+D+i project TRAINER-A (ref.PID2020-118011GB-C21), funded by MCIN/ AEI/, Spain10.13039/501100011033. This work is also partially funded by the Catalan Institution for Research and Advanced Studies (ICREA), Spain and the Secretariat for Universities and Research of the Ministry of Business and Knowledge of the Government of Catalonia, Spain and the European Social Fund.

* Corresponding author.
E-mail address: miquel.ferriol@upc.edu (M. Ferriol-Galmés).

https://doi.org/10.1016/j.comnet.2022.109329
Received 22 December 2021; Received in revised form 20 July 2022; Accepted 25 August 2022
Available online 3 September 2022

including traffic multiplexing and demultiplexing along with queues. In addition, it imposes strong assumptions on packet arrivals, which typically do not hold in real networks [13].

Second, fluid models have become a popular alternative for network optimization [11,14–17], as they are simple but practical in some cases. Indeed, fluid models are useful for several optimization tasks, such as balancing link utilization. Nevertheless, fluid models assume constant per-link delays and do not consider the effects of queuing delays, scheduling policies, and network losses. Therefore, they offer limited accuracy in networks operating at high utilization regimes or with complex queuing policies. We provide experimental evidence of this in Section 3.

Third, packet-level network simulators are arguably among the most accurate alternatives to traditional network models. However, these simulators suffer from a high computational cost, as they rely on simulating individual packet events. This often makes them unable to handle real-world scenarios with large traffic volumes, and to operate at short time scales (e.g., real-time QoS inference [18]).

Recent advances in Machine Learning (ML) [19] have led to a new set of efficient techniques that can be leveraged in network modeling. Particularly, Deep Learning (DL) looks very promising for this purpose [20]. The main advantage of DL models is that they are *data-driven*, that is, they are trained with real-world data, achieving unprecedented accuracy by effectively modeling the entire range of complex network characteristics. Given its ambition, those models are commonly referred to as *Digital Twins* [21–24]. Existing DL-based solutions for network modeling [25,26] mainly rely on classic Neural Network (NN) architectures, such as MultiLayer Perceptron (MLP), or Recurrent Neural Networks (RNN). However, computer networks are intrinsically represented in a graph-structured manner, and classic NNs are not suited to learn from graphs. In this context, Graph Neural Networks (GNN) [27] have recently emerged as an effective technique to model graph-structured data. Particularly, these new types of neural networks are tailored to understand the complex relationships between the elements of a graph. More in detail, the internal architecture of a GNN is dynamically built based on the elements and connections of input graphs, and this allows to learn generic modeling functions that do not depend on the graph structure. Please note that Digital Twins can be built using different Machine Learning techniques other than Deep Learning.

In this paper, we present TwinNet, a Digital Twin that models the complex relationship between topology, routing, queue scheduling, and input traffic, in order to produce accurate estimates of per-flow QoS metrics (e.g., delay, jitter, loss). TwinNet is able to accurately estimate the delay in paths traversing arbitrary concatenations of queuing policies, with different routing configurations, traffic matrices, and network topologies. A critical feature of TwinNet is its ability to *generalize* to unseen networks. This means that it can provide accurate estimates in networks with different characteristics to those seen in training. We refer the reader to Section 7.4 for a detailed analysis of this aspect.

The following paragraphs summarize the main contributions of this paper:

Digital Twin: We propose TwinNet, a Digital Twin that can model networks with arbitrary concatenations of queuing configurations (with various scheduling policies, number of queues, and queue sizes), source–destination routings, topologies, and traffic models. We train TwinNet with a *packet-accurate* network simulator using an extensive dataset with several real-world network topologies. TwinNet shows a worst-case Mean Absolute Percentage Error (MAPE) of 3.88%. We also validate TwinNet with real-world packet traces, obtaining a MAPE of 7.2% and with data from a real testbed observing a MAPE of 6.3%.

Generalization: A common downside of ML solutions is their poor performance when operating in networks different from those seen during the training phase, which is referred to as lack of *generalization* [28]. Without generalization, training *must* be done at the same network where the ML-based solution is expected to operate. However, this is not practical in production networks, because the ML model needs to observe link failures, packet loss, etc. As expected, this is not desirable in a production network. Hence, from a practical standpoint, generalization is a crucial capability. TwinNet is capable of generalizing to unseen scenarios by changing its internal structure based on the input topology graph. We provide experimental evidence of the generalization capabilities of the model. Specifically, we evaluate TwinNet with 106 real-world networks (never seen in training) from the Internet Topology Zoo [29] and observe that the model achieves a MAPE of 4.5%.

Benchmark against the state-of-the-art: We benchmark TwinNet against a Multi-Layer Perceptron as described in *F. Krasniqi* et al. [30], and a well-established standard Graph Neural Network architecture [31]. Our results show that TwinNet outperforms both solutions.

Optimization: Lastly, we pair TwinNet with an optimizer to find which routing *and/or* scheduling policies fulfill complex SLAs, with increasing traffic intensity. Our results show that, even in highly congested scenarios, TwinNet manages to satisfy the imposed SLAs, while state-of-the-art techniques based on fluid models cannot guarantee them in the same scenarios.

The remainder of this paper is structured as follows: First, in Section 2 the considered network scenario is introduced. Second, Section 3 explores the limitations of current State-of-the-Art optimizers. In Section 4 the idea of Graph Neural Network is introduced and its internal Message-Passing architecture is explained. Afterwards, Section 5 describes the proposed GNN-based network model, as well as its internal Message Passing architecture. Section 6 talks about how the prototype has been implemented and which hyperparameters have been used. Section 7 evaluates TwinNet in four different simulated scenarios. Then, Section 8 presents several relevant use cases that illustrate the potential of TwinNet when paired with a network optimizer as well as one scenario with a real testbed that re-evaluates the accuracy of TwinNet. Finally, Sections 9 and 10 present the related work and the conclusions respectively.

2. Network scenario

Novel network applications (Industry 4.0 [5], Tactile Internet [6], etc.), are pushing further the requirements of network-offered SLAs. At the time of this writing, there are substantial research efforts to accommodate such challenging SLAs with new protocols and architectures [2,8,9]. Many of such architectures take advantage of the Software-Defined Networking (SDN) paradigm, which enables a new breed of centralized optimization algorithms [11]. Centralization enables full visibility of the network configuration and state, as well as fine-grained flow control.

In this paper, we consider a Wide Area Network (WAN) that implements a classical SDN architecture: a centralized controller, and a southbound protocol that allows configuring the devices and collecting network performance metrics, such as OpenFlow or NETCONF (Fig. 1). The SDN controller has an SLA/QoS optimizer application, that guarantees fine-grained SLAs by adjusting the routing and queuing policies according to the network state. First, the network administrator defines the desired set of SLAs, e.g. the *maximum* mean delay of flows. The granularity of the flows depends on the requirements of the administrator, hence, we consider different flow granularities, ranging from source–destination to 5-tuple flows. Then, SLAs are mapped to the data plane by tagging the packets of each flow using common fields from the IP header: Type-of-Service (ToS) for IPv4, and Differentiated Services for IPv6.

Second, a network model – TwinNet in this case – is paired with an optimizer (see Section 8 for further details) running on the SDN controller. The controller has visibility of the local configuration of each data-plane element as well as up-to-date measurements of the network state: bandwidth, mean delay of each source–destination pair, and

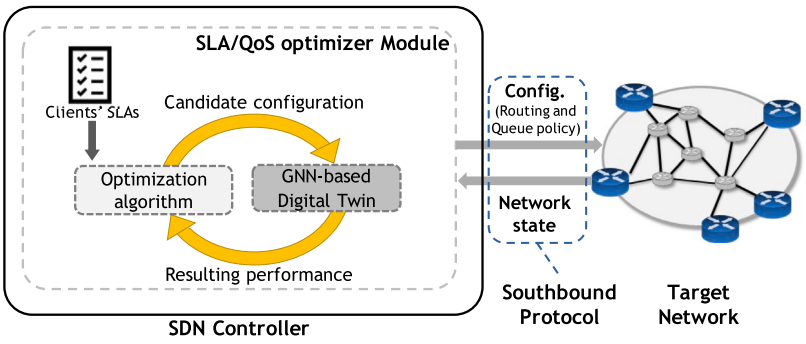


Fig. 1. Network scenario for SLA-based optimization.

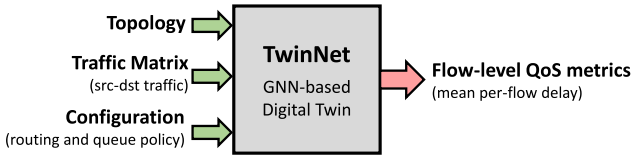


Fig. 2. Blackbox representation of TwinNet.

link utilization. This can be achieved using readily available telemetry methods [32,33].

Leveraging this information, the optimizer explores alternative configurations that can meet the SLA for the current traffic load. In TwinNet, configurations are combinations of source–destination routing and per-interface queuing configurations, i.e., scheduling algorithm and queue parameters. Particularly, in this paper, we consider the following queue scheduling policies: First In First Out (FIFO), Strict Priority (SP), Weighted Fair Queuing (WFQ), and Deficit Round Robin (DRR). Each configuration produced by the optimizer is tested by TwinNet, which produces fast and accurate estimates of flow delays. Once the optimizer finds a configuration that meets the SLAs, it is applied to the data-plane elements.

Thus, our Digital Twin must satisfy three main requirements: (*i*) Generate accurate predictions of SLA metrics, (*ii*) Achieve fast operation, to quickly adapt the configuration to traffic changes, and (*iii*) *Generalize* to other network scenarios not seen during training. While network packet-level simulators are accurate, they are slow since they need to simulate the forwarding, transmission, and propagation of each and every packet. On the other hand, traditional analytical models such as queuing theory are fast but not accurate in the presence of non-markovian traffic [34]. Finally and as we will see in the paper, many Deep Learning architectures fail to generalize to scenarios not seen in training.

The latter feature is of critical importance. In the context of computer networks, training a Digital Twin requires generating a large diversity of network scenarios (e.g., random routing and queuing configurations, simulating link failures, etc.), which could render the network unusable. This would be infeasible in production networks. Thus, we argue that a practical way to build Network Digital Twins is training them in controlled environments (e.g., in a networking lab). Then, they can take advantage of their generalization capabilities to operate in real network topologies unseen in advance. This is one of the main limitations of existing network models based on traditional neural networks (e.g., Multi-layer Perceptron, Convolutional Neural Networks), as they do not generalize well to other networks. Hence, they need to be trained directly in the production network (where it is not admissible to generate wrong configurations) or at least be re-adjusted using transfer learning [35].

TwinNet exploits its internal Graph Neural Network (GNN) architecture to model the complex relationships between the various components that define the network state, in order to predict the global QoS. Particularly, the proposed Digital Twin (Fig. 2) is fed with a network state snapshot, defined by: (*i*) a network topology, (*ii*) a src-dst traffic matrix, and (*iii*) a routing and queueing policy. As output, it produces estimated per-flow performance metrics. Note that in this paper we train this model to predict the mean per-flow delay.

3. Limitations of state-of-the-art optimizers

State-of-the-art network optimization solutions mainly rely on fluid models [11,14–17]. In order to discuss the limitations of such models when dealing with complex SLA scenarios, we take DEFO [14] as a representative of the state-of-the-art in this area [36].

DEFO is a constraint programming framework for network optimization. This optimizer allows network administrators to set constraints to the optimization problem, including maximum end-to-end delays. To estimate the performance of candidate configurations, DEFO uses a fluid model of the network. In DEFO, the per-link delay is a fixed input variable of the network optimizer. Particularly, this value is either provided by the network operator in real topologies, or computed manually according to the link distance in synthetic topologies [14]. Then, the delay of a path is assumed to be equal to the sum of transmission delays of the links that it traverses, without considering any queuing delay.

To test the accuracy of the fluid model used by DEFO, the actual delay is measured with an accurate packet-level simulator based on OMNET++ [37]. In this particular case, the simulation of the network scenario is defined by a topology, a src-dst traffic matrix, and a routing configuration. In particular, we use three different real-world topologies (NSFNET [38], GEANT [39] and GBN [40]). The traffic matrix is defined following the same approach as described in [31], generating three network scenarios containing 0%, 0.8%, and 1.3% off packet losses to see the effect of the network congestion on the accuracy of the fluid model. Finally, the routing configurations are the ones returned by DEFO after running its optimization process.

Fig. 3 plots the Cumulative Distribution Function (CDF) of the relative error produced by the fluid model when estimating the per-path delay in a network scenario from [14], optimized with DEFO. The figure plots three different distributions according to the network load (average packet loss from 0% to 1.3%). These results show that actual delays are different from those estimated by the fluid model. Even for scenarios without packet loss, the fluid model has a Mean Absolute Percentage Error (MAPE) of 21%, while estimations degrade significantly with increasing network load (≈50% MAPE). This is due to the fact that fluid models do not consider the queuing delay, which becomes an important component of the end-to-end delay in the presence of congestion.

Finally, it is worth noting that DEFO is just used in this section to illustrate the limitations of fluid models for SLA-aware optimization, but that contributions in [14] go well beyond the network model used for optimization. Indeed, the DEFO optimizer could easily support more complex network models, like the one proposed in this paper.

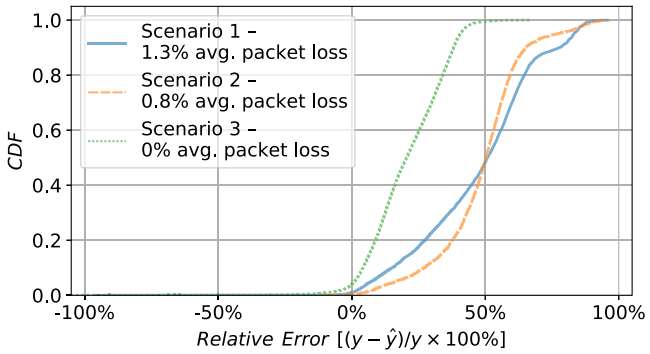


Fig. 3. CDF of the relative error of the fluid model under various traffic loads.

4. Background on Graph Neural Networks

Graphs are a fundamental data type used to represent relational information. Particularly, a graph $G \in \{V, E\}$ is defined by a set of objects V (*vertices*) with some relationships between them E (*edges*). GNNs [27] are a recent family of Neural Networks (NN) especially conceived to work with graph-structured data. These models dynamically build their internal architecture based on the input graph. They use a modular NN structure that explicitly represents the elements and connections of the input graph. As a result, they support graphs of variable size and structure. Moreover, their graph processing mechanism is invariant to node and edge permutation, which eventually endows them with strong generalization capabilities over graphs (a.k.a., strong relational inductive bias [41]).

GNNs have already been successfully applied to many different problems where data is fundamentally represented as graphs, such as prediction of molecular properties in chemistry [42], modeling of complex gravitational systems in physics [43], deciphering protein interactions in biology [44], or recommendation systems for social networks [45].

Despite GNNs comprise a broad class of neural network models for graphs with different architectural variants (e.g., [27,43,46]), all of them share a basic principle: first, an iterative message-passing phase, and then a readout phase that produces the output of the model. The following subsections elaborate on the different modules that shape a basic GNN, including all the functions these modules implement internally.

4.1. Message-passing architecture

As mentioned earlier, a key feature that distinguishes GNNs from other well-known NN families is that the internal NN architecture is not fixed, but depends on the structure of the input graph. In a GNN, each graph element is represented with a *hidden state* h_i (encoded as an n-element vector initialized with some features), and hidden states are combined according to the graph connections, through an iterative message-passing process. During each message-passing iteration the hidden state of each node is updated with the states received from its direct neighbors, thus the information is gradually propagated over the whole graph after several iterations. Each node maintains a specific representation of the graph based on its local perspective, since it only receives information from its neighbors. As a reference, the information is potentially propagated over the full graph when the number of message-passing iterations is equal to the graph diameter (i.e., number of hops between the two most distant elements in the graph).

Specifically, a GNN is comprised of two main modules: (*i*) Message-passing, and (*ii*) Readout.

Message-passing module

This first module executes an iterative message passing over the graph elements. It can be executed for a fixed number of iterations T, or until a convergence criterion is satisfied [27]. In general, each message-passing iteration can be defined by the following equations:

$$M_{ij}^{*} = m(h_i^t, h_j^t, e_{ij}) \tag{1}$$

$$M_i^{t+1} = aggr(M_{ij}^{*}) \tag{2}$$

$$h_i^{t+1} = u(h_i^t, M_i^{t+1}) \tag{3}$$

In other words, the message-passing module runs three different functions that are executed over all the graph elements:

1. Message function $[m_{ij}(h_i, h_j, e_{ij})]$: This function encodes information about the relation of two connected elements in the graph. It takes as input the hidden states of two nodes (h_i and h_j), and optionally a vector describing some properties of their relation (e_{ij}). Its output is a *message* (n-element vector) that encodes information about this relation (M_{ij}^{*}).
2. Aggregation function $[aggr(M_{ij}^{*})]$: This function aggregates all the messages computed for each element with its neighbors so that the result is a fixed-size vector (M_i) that encodes information about the local neighborhood of the node. The main objective of this aggregation function is to keep a fixed-size output representation combining all the messages received by each node, independently of the number of neighbors. This enables to handle an arbitrary number of connections per node (i.e., graphs with arbitrary node degrees). Typically, this function is implemented with an element-wise summation, although other aggregation functions such as mean, min, max, variance, or combinations of them can be used [47].
3. Update function $[u(h_i^t, M_i^{t+1})]$: Lastly, the hidden states of each node (h_i^t) are combined with the aggregated message computed for the node (M_i), producing an updated hidden state representation (h_i^{t+1}). This new hidden state is then used as input to the next message-passing iteration — executing again Eqs. (1)–(3).

In this module, the Message (m) and Update (u) functions are implemented with two independent neural networks (typically fully connected) whose internal parameters (i.e., neuron weights and biases) are learned during the training phase.

Readout module

Once the message passing has finished, the resulting node hidden states (h_i) are passed through the readout module. In this last phase, the information encoded in the hidden states is converted to the output values or labels of the GNN $[\hat{y}=r(h_i|v \in V)]$. Note that GNNs may typically produce two different output types: (*i*) node-level, or (*ii*) global graph-level labels. In the former case, the outputs are obtained by applying a readout function (r_n) individually on each node hidden state, while global outputs are computed by first aggregating all the hidden states and then applying a global readout function over the resulting aggregated vector (r_G). Similar to the message-passing module, the aggregation function allows the GNN to support graphs of arbitrary size, as it produces a fixed-size vector regardless of the number of nodes in the input graph. Readout functions (r_n or r_G) are also approximated by a neural network (e.g., fully connected), while the aggregation function is often implemented as an element-wise summation.

At this point, it is important to note that a basic GNN typically comprises four unique functions (m, $aggr$ and u, r) that are replicated and combined according to the input graph, so *the same Message, Aggregation, Update and Readout functions are applied to all the graph elements*. During training, the functions implemented by NNs (m, u, and r) are gradually approximated by applying a common backpropagation algorithm [48] over the whole GNN architecture, considering the

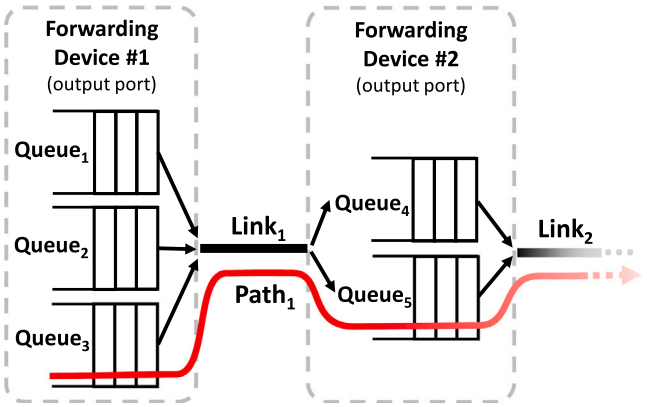


Fig. 4. Schematic representation of the network model implemented by TwinNet.

output labels of the GNN and those of the training dataset (i.e., end-to-end training). The key aspect of this process is that, for each training sample, the NN parameters are jointly optimized across all the instances of the NN within the GNN model. This process builds generic *Message*, *Update* and *Readout* functions jointly learned from the local perspective of each individual node in the graph. Then, these generic functions can be applied to other graphs unseen in advance, as long as they are used for the same purpose they were trained for, e.g. predicting path delays in our case.

5. TwinNet: Digital Twin with GNN

In this section, we describe TwinNet, a GNN-based Digital Twin for SLA modeling and optimization. This model accurately infers the impact of different routing and queueing configurations (scheduling algorithm and queue parameters) on network performance. Likewise, it is tailored to generalize to different network topologies and traffic intensities not seen before.

5.1. Overview

TwinNet implements a novel and custom GNN architecture inspired by the inherent behavior of computer networks, where there are different components (e.g., forwarding devices, configuration, traffic) that interact with each other and have a complex non-linear impact on network performance.

The main intuition behind this architecture is as follows. The model considers an input graph with three main network components: (*i*) the links that shape the network topology, i.e. connections between network devices, (*ii*) the queues on each output port of network devices, and (*iii*) the src-dst paths resulting from the input routing configuration. Each of these elements is explicitly represented in the GNN with n-element vectors that encode their hidden states (h_l, h_q, and h_p respectively). They are combined through a message-passing algorithm that aims to capture the relation between the topology, traffic, routing and queueing policy of the input network scenario. Fig. 4 represents how TwinNet models these three components. First, the state of paths is affected by the concatenation of the queues and the links they traverse. For instance, in Fig. 4, $path_1$ follows the sequence: $[queue_3, link_1, queue_5, link_2 ...]$. At the same time, the state of queues and links depends on all the paths passing through them. Hence, there is a circular dependency between the states of paths, links, and queues that the GNN model must resolve to eventually produce accurate per-path QoS estimates. Note that we assume that the forwarding engine of network devices is constant and ideal, hence it does not introduce any other potential hardware-level effects on the delay of queues.

5.2. Notation

We define the network topology as a set of links $L = \{l_i : i \in (1, \dots, n_l)\}$ and the queues on output ports of network devices $Q = \{q_i : i \in (1, \dots, n_q)\}$ and a set of source–destination paths $P = \{p_i : i \in (1, \dots, n_p)\}$. Let us also consider a path as a sequence of tuples with the queues and links they traverse defined by the routing scheme. Hence, we define the paths as: $p_i = \{(q_{PQ(p_i,0)}, l_{PL(p_i,0)}), \dots, (q_{PQ(p_i,|p_i|)}, l_{PL(p_i,|p_i|)})\}$, where $PQ(p_i, j)$ and $PL(p_i, j)$ respectively return the index of the jth queue or link along the path p_i. Let us also define $Q_p(q_i)$ as a function that returns all the paths passing through queue q_i, and $L_q(l_i)$ as a function that returns the queues injecting traffic into link l_i — i.e., the queues at the output port to which the link is connected.

Each queue, link and path are initialized with some features x_{l_i}, x_{q_j} and x_{p_k}, respectively. In our particular case, we set the initial features of links (x_l) as (*i*) the link capacity, and (*ii*) the scheduling policy (FIFO, Strict Priority, WFQ, or DRR) configured in the egress port that injects traffic into the link, using one-hot encoding. The initial features of queues (x_q) are: (*i*) buffer size, (*ii*) priority order (one-hot encoding), and (*iii*) weight (only for WFQ and DRR). Lastly, we set the initial path features (x_p) as the traffic volume (bits and packets) sent from the source to the destination node of the path.

5.3. Network model

We initialize the state of links h_l, queues h_q, and paths h_p respectively with their initial feature vectors (x_l, x_q and x_p), and apply zero-padding to fit the size of the target vectors, which is a configurable parameter of the GNN. After the message-passing phase, these hidden states are expected to encode some meaningful information about links (e.g., utilization), queues (e.g., load, packet loss rate), and paths (e.g., end-to-end delay, packet loss) based on the information exchanged along the graph. Thus, TwinNet is based on these basic principles:

1. The state of a path depends on the states of all the queues and links that it traverses.
2. The state of a link depends on the states of all the queues that inject traffic into the link.
3. The state of a queue depends on the states of all the paths that inject traffic into the queue.

These principles can be mathematically formulated as follows:

$$h_{q_i} = f_q(h_{p_1}, \dots, h_{p_m}), \quad q_i \in p_k, k = 1, \dots, j \tag{4}$$

$$h_{l_j} = f_l(h_{q_1}, \dots, h_{q_m}), \quad q_m \in L_q(l_j) \tag{5}$$

$$h_{p_k} = f_p(h_{q_{k(0)}}, h_{l_{k(0)}}, \dots, h_{q_{k(|f_k|)}}, h_{l_{k(|f_k|)}}) \tag{6}$$

where f_q, f_l, and f_p are some unknown functions.

A direct approximation of functions f_q, f_l and f_p is complex given that: (*i*) Eqs. (4)–(6) define a complex non-linear system of equations with the states being hidden variables, (*ii*) they encode complex mutual dependencies between different network components (topology, routing, queueing policies, traffic), and (*iii*) the dimensionality of possible states is extremely large.

GNNs have shown an outstanding capability to work as universal approximators over graph-structured data [41,49]. As a result, thanks to its internal GNN-based architecture, TwinNet is able to approximate flexible f_q, f_l and, f_p functions, which can later be applied to unseen topologies, routing schemes, queueing configurations, and traffic distributions.

Algorithm 1 Internal architecture of TwinNet

Input: L, Q, P, x_q, x_l, x_p
Output: h_q^T, h_l^T, h_p^T, $\hat{y}_p$

1: **for each** $l \in L$ **do** $h_l^0 \leftarrow [x_l, 0...0]$
2: **for each** $q \in Q$ **do** $h_q^0 \leftarrow [x_q, 0...0]$
3: **for each** $p \in P$ **do** $h_p^0 \leftarrow [x_p, 0...0]$
4: **for** t = 0 to T-1 **do**
5: **for each** $p \in P$ **do**
6: **for each** $q, l \in p$ **do**
7: $h_p^t \leftarrow RNN_p(h_p^t, [h_q^t, h_l^t])$
8: $\widetilde{m}_p^{t+1} \leftarrow h_p^t$
9: $h_p^{t+1} \leftarrow \widetilde{h}_p^t$
10: **for each** $q \in Q$ **do**
11: $M_q^{t+1} \leftarrow \sum_{p \in Q_p(q)} \widetilde{m}_{p,q}^{t+1}$
12: $h_q^{t+1} \leftarrow U_q(h_q^t, \widetilde{M}_q^{t+1})$
13: $\widetilde{m}_q^{t+1} \leftarrow h_q^{t+1}$
14: **for each** $l \in L$ **do**
15: **for each** $q \in L_q(l)$ **do**
16: $h_l^t \leftarrow RNN_l(h_l^t, \widetilde{m}_q^{t+1})$
17: $h_l^{t+1} \leftarrow h_l^t$
18: $\hat{y}_p \leftarrow F_p(h_p)$

5.4. Proposed GNN architecture

Algorithm 1 describes the internal architecture of TwinNet. This custom GNN architecture is especially designed to solve the circular dependencies described in Eqs. (4), (5) and (6) by executing an iterative message-passing process. First, the hidden states h_l, h_q, and h_p are initialized (lines 1–3) using x_l, x_q, and x_p respectively and padded with zeros to the specific hidden state dimension. After the hidden state initialization, the message passing phase begins. During this step, each state is combined with their connected elements according to the relations described in the input graph. This process is repeated T iterations (loop from line 4). Thus, by the end of the message-passing execution, hidden states h_l, h_q, and h_p should eventually converge to some fixed values after exchanging information with their neighbors in the graph [27].

Unlike standard GNN models (see Section 4), TwinNet implements a more complex message passing that can be divided into three different stages involving message exchanges between heterogeneous graph elements. The loops from line 5, line 10, and line 14 in Algorithm 1 represent these different message-passing stages, where for each path (line 5), for each queue (line 10), and for each link (line 5), the hidden states are exchanged with their connected elements and updated based on the information received. More specifically, each path collects messages from all the queues and links that it crosses (loop from line 6), then each queue receives messages from all the paths that pass through it (summation from line 11) and lastly, each link collects information from all the queues that inject traffic into it (loop from line 15). To aggregate the paths' hidden states on queues (line 11) we use an element-wise summation. In the case of links and queues, it is important to consider that there is a sequential dependence on the elements connected. For instance, the order of queues and links that a path traverses is important in case there is packet loss, as the packets dropped by one queue will not be injected into the subsequent links and queues. For this reason, we use a Recurrent Neural Network (RNN) to aggregate the sequences of queues and links on the paths' hidden states (line 7). Similarly, the model aggregates the queue states on their related links using another RNN (line 16), as it is important to maintain the order of queues to model the behavior of the queuing policy (e.g., the priority order). For simplicity of the GNN architecture, we implement some message and update functions as direct variable assignments, except for the case of the update function for queues U_q (line 12), which is implemented using a Gated Recurrent Unit to facilitate the convergence of the algorithm [50].

Finally, the function F_p (line 18) represents a readout function that is applied individually on each path hidden state h_p and, in this case, is used to finally produce the estimated mean per-path delays ($\hat{y}_p$). Particularly, we modeled the readout function F_p with a fully connected NN using a SELU activation function [51].

This architecture provides flexibility to represent any routing configuration and queuing policy (including QoS-aware scheduling algorithms with multiple queues). This is achieved by the direct mapping of the paths resulting from the routing configuration P to specific message passing operations with queues, links, and other paths.

6. Prototype implementation

We implemented a prototype of the full TwinNet message-passing structure using TensorFlow. The source code of TwinNet and the training/evaluation datasets used in this paper are publicly available at [52].

6.1. Simulation setup

In order to train our GNN-based Digital Twin, we built the ground truth leveraging a packet-level network simulator (OMNeT++ v5.5.1 [37]). Each sample in the training set corresponds to the simulation of a specific network scenario, defined by a topology, a src-dst traffic matrix, and a routing and queuing policy. Then, the simulator labels this sample with the mean per-packet delay measured on each source–destination path. Regarding the training dataset, each sample represents a random selection of input features (topology, traffic, routing, and queuing configuration) according to the following descriptions:

6.1.1. Traffic

We generate the input Traffic Matrices (TM) following the same approach described in [31]. In our particular case, traffic matrices are generated to cover a wide range of operational scenarios from low traffic loads to highly loaded networks. Depending on the capacities of the links, as well as the routing configuration of the network, this TM will result in a certain packet loss. Particularly, we generated these TMs to obtain a maximum packet loss of 3% according to [53]. Since some TMs lead to traffic aggregates that exceed the capacity on some links, they will cause congestion and packet loss due to the accumulation of packets in the queues. Note that we also use traffic extracted from real packet traces, which is later described in Section 7.5. Finally, we randomly assign a Type-of-Service (ToS) label to each source–destination traffic flow (ToS∈[0–9]), which is then used to map traffic flows to specific queues at egress ports of network devices — as shown below.

6.1.2. Queueing configuration

Each node is configured randomly with: (*i*) a queue scheduling policy, that can be First In First Out (FIFO), Strict Priority, Weighted Fair Queueing (WFQ) or Deficit Round Robin (DRR), (*ii*) a random number of queues, and (*iii*) a random queue size. For WFQ and DRR, we define a set of random queue weights. Finally, we map ToS labels to queues in decreasing order of priority, including a random component and depending on the number of queues. Hence, lower ToS labels are assigned to higher priority queues. Note that the dataset contains samples of a wide range of queuing configurations from the simulator, this helps the GNN model understand their effect on network performance.

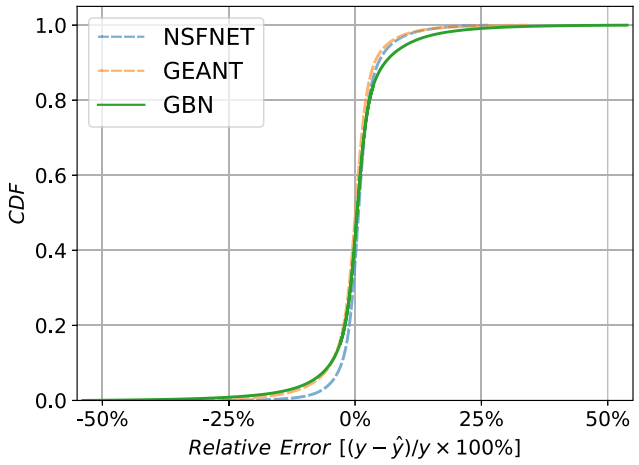


Fig. 5. CDF of relative error. y represents the true delay, while $\hat{y}$ denotes the predicted one.

6.1.3. Topologies

In order to train and evaluate the model, we use three different real-world topologies (NSFNET [38], GEANT [39] and GBN [40]). Later, in Section 7.4, we also evaluate the generalization properties of our solution with 106 real-world topologies from the Internet Topology Zoo [29].

6.2. Machine learning framework

We train the model using 100k samples from each training network (NSFNET and GEANT). Note that for a given network topology, a data sample is defined as a random combination of routing, queuing policy, and traffic matrix. The randomly generated configurations are within the operational ranges defined in Section 6.1.

Our model has two relevant hyper-parameters that can be fine-tuned: (i) The size of the hidden states h_l, h_q and h_p, and (ii) the number of message-passing iterations (T). Based on preliminary experiments we use 32-element vectors for all the hidden states, and $T = 8$ iterations. In this context, T should correlate to the network diameter, which in real networks approximately scales with $log(N)$, where N is the number of nodes of the input graph [54].

We choose the Mean Squared Error (MSE) as a loss function, which is minimized using an Adam optimizer with an initial learning rate of 0.001 and a decay rate of 0.6 executed every 80,000 steps. In addition to this, we added a $L2$ regularization loss of λ=0.1.

Fig. 5 shows the CDF of the relative error when predicting the mean per-path delay for the three topologies. In the two topologies seen in training (NSFNET and GEANT), TwinNet obtains a MAPE of 2.59% and 3.01% respectively. Int the GBN topology, only used in the evaluation, obtains a $MAPE = 3.88\%$. Also, the error distribution is centered around 0, which means that the model predictions are not biased towards under- or overshooting.

7. TwinNet evaluation

Our evaluation focuses on several relevant properties of TwinNet including: (i) the accuracy of the delay prediction of source–destination pairs, in a wide variety of topologies, routing, queueing configurations, and traffic matrices with various load levels, (ii) the generalization capabilities of the model in networks never seen during training, (iii) the accuracy when working with real-world data, and (iv) the speed and the scalability of its estimations.

7.1. Baselines

To benchmark the performance of TwinNet we compare it against two state-of-the-art ML-based models. The first one, based on the work described at [30], where the per-path delay is modeled using a multi-layer perceptron (MLP). In this particular case, the model takes as input the traffic matrix (TM) and the queue and link configuration, similarly to the ones described in 5.4. In our particular case, the MLP consists of four layers (an input, an output, and two hidden layers) of nonlinearly-activating nodes. Each node in each layer is connected to every node in the following layer via a certain weight w_{ij}. These weights (w_{ij}) are updated during the training phase based on the amount of error in the output compared to the expected result. The second, RouteNet [31] is a GNN-based model specially designed to model networks. Similar to TwinNet, RouteNet models the network by performing a two-stage message passing between the links and paths in the network.

Table 1
Comparison of performance metrics.

(a) Performance comparison for NSFNET and GEANT networks, seen during training.

	NSFNET				GEANT			
	MAPE	MSE	MAE	R^2	MAPE	MSE	MAE	R^2
MLP	0.46	4.23	0.779	−0.158	0.358	1.114	0.303	−0.039
RouteNet	0.667	4.366	0.847	−0.193	0.581	1.172	0.336	−0.093
TwinNet	**0.033**	**0.077**	**0.06**	**0.978**	**0.039**	**0.028**	**0.034**	**0.973**

(b) Performance comparison for GBN network, never seen during training.

	GBN			
	MAPE	MSE	MAE	R^2
MLP	0.874	3.093	0.537	−0.078
RouteNet	0.533	3.124	0.523	−0.089
TwinNet	**0.034**	**0.067**	**0.046**	**0.976**

7.2. Performance metrics

Performance metrics are a critical component of the evaluation frameworks in Machine Learning. They are mainly used to monitor and measure the performance of a model. Since we are in a regression problem and following the approach described in [55], we provide 4 different metrics. Two absolute metrics: Mean Squared Error (MSE) and Mean Absolute Error (MAE), as well as two relative metrics: Mean Absolute Percentage Error (MAPE) and the Coefficient of Determination (R^2).

We believe that combining the four of them provides a good picture of the performance of the different models evaluated in the following sections. Mainly, we focus on MAPE as, in contrast to MAE and MSE, it is a relative metric that does not depend on the units of the predicted variable (delay).

7.3. Accuracy

We evaluate the accuracy of TwinNet and both baselines using 100k samples of each of the aforementioned topologies (NSFNET, GEANT, and GBN). Note that the GBN topology has never been seen during the training phase for any one of the models.

Table 1 shows the results for the three topologies. In this particular case, we show the Mean Absolute Percentage Error (MAPE), the Mean Squared Error (MSE), the Mean Absolute Error (MAE), and the Coefficient of determination (R^2). As we can see, TwinNet clearly outperforms both baselines, achieving a MAPE of 3.8% and an R^2 of 0.976 for the GBN topology.

7.4. Generalization capabilities

GNN models have shown a great potential to generalize over data structured as graphs [41,49]. This section presents an analysis of the generalization capabilities of TwinNet. Particularly, we refer to generalization as the capability of the model to make accurate predictions in

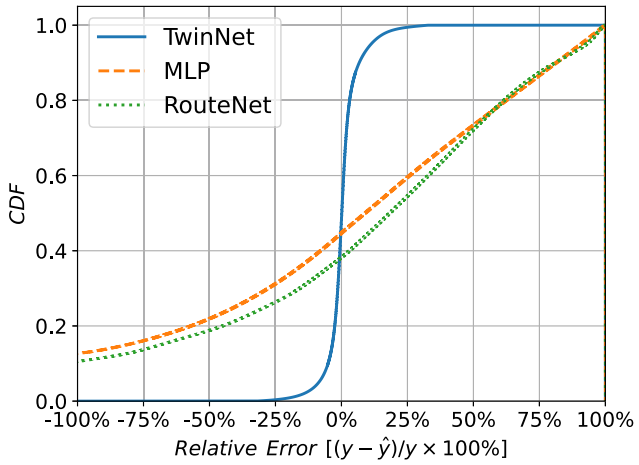


Fig. 6. CDF of relative error over 106 unseen real-world topologies.

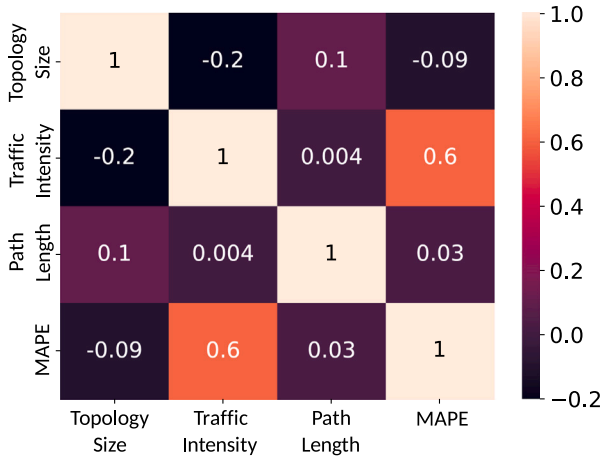


Fig. 7. Correlation matrix of the topology size, traffic intensity, path length, and MAPE.

Table 2
Performance metrics comparison over 106 real-world topologies never seen during training.

	TOPOLOGY ZOO			
	MAPE	MSE	MAE	R^2
MLP	0.852	16.864	0.336	−0.174
RouteNet	0.643	14.44	0.322	−0.006
TwinNet	**0.038**	**0.476**	**0.0258**	**0.966**

Table 3
Performance metrics comparison over 106 real-world topologies with realistic traffic never seen during training.

	TOPOLOGY ZOO WITH REALISTIC TRAFFIC			
	MAPE	MSE	MAE	R^2
MLP	0.758	13.32	0.299	0.071
RouteNet	0.686	14.48	0.351	−0.008
TwinNet	**0.071**	**0.104**	**0.028**	**0.992**

new network scenarios unseen during the training phase. In our case, it involves different topologies, routing and queuing configurations, and traffic distributions never seen during training.

To this end, we evaluate the accuracy of the proposed GNN-based model as well as both baselines, with 106 real-world topologies from the Internet Topology Zoo [29] that were not present in the training set. For each topology, we use the network simulator previously described (Section 6.1) to generate the delay labels for the ground truth, considering a variety of traffic matrices and configurations. Then we analyze the accuracy of the model trained only with the NSFNET and GEANT topologies (Section 6.2).

Fig. 6 shows the CDF of the relative error for this experiment. As the figure shows, our model provides good generalization capabilities, achieving a $MAPE = 3.8\%$ over the 106 real-world topologies never seen during training. As expected, both baselines perform poorly in generalization scenarios with a MAPE greater than 70%.

Table 2 shows a summary of the different metrics evaluated on the 106 real-world topologies obtained from the Internet Topology Zoo. Again, TwinNet outperforms both baselines, which perform poorly when the topologies evaluated are different from the ones seen during training.

We have experimentally analyzed what features have more impact on the model's accuracy, and have found that there is little variability in the error across the different topologies (≈0.8% MAPE between the topologies with less and more error). Fig. 7, shows how the error correlates with the traffic intensity. This is in line with the accuracy results we obtained in previous experiments where the model showed slightly less accuracy in samples from highly congested networks. As an example, networks with very low traffic obtain an average error of 1.5%, while networks with a high level of congestion produce an error of 5.3% on average.

7.5. Experiments with real traffic

In the previous experiments, we evaluated TwinNet with synthetic traffic. In this section, we validate the accuracy of this model when applied to real traffic, without retraining the model.

For this purpose, we use real-world traffic matrices from the SNDlib library [56]. Since the traffic matrices only contain traffic aggregates of each source–destination pair, we use a recent snapshot from the MAWI repository (SamplePoint-F, Oct. 2020) [57] to extract realistic packet inter-arrival times. Then, we scale these inter-arrivals according to the values in the traffic matrices. Regarding the mapping of source–destination flows to ToS classes, we follow the same distribution from a real ISP [58].

We create a new dataset only for evaluation, not training. This dataset contains three different topologies: GBN (NOBEL), GEANT, and ABILENE, extracted from SNDlib [56], and the aforementioned traffic matrices. Note that the dataset contains: (*i*) two new topologies (GBN and ABILENE), not present in the training dataset, and (*ii*) traffic matrices completely different from the ones used in training. We evaluate the accuracy of the previous model from Section 7.3 directly in this dataset, without retraining it.

Fig. 8 shows the CDF of the relative error in this new scenario. As we can observe, TwinNet produces accurate delay estimates even in scenarios emulating real traffic. Particularly, the model achieves a MAPE of 5.6% for the ABILENE topology, 7.1% for GEANT, and 8.9% for GBN. As mentioned previously, the model has been trained with synthetic traffic, and here we test it using real traffic. Despite the traffic seen by the model following a slightly different traffic distribution than that seen during training, the model still achieves good accuracy. Compared with the results from Fig. 5, the MAPE increases from 3.9% (results with synthetic traffic) to 5.6%–8.9% MAPE (best and worst case results with real traffic).

Finally, we evaluate TwinNet using a dataset that combines the 106 real network topologies previously used (Section 7.4), and with traffic matrices that follow the inter-arrivals times and ToS classes mentioned used in the previous experiment [57,58]. Fig. 9 and Table 3 show the performance of TwinNet with respect to the baselines. Following the trend of the previous experiments, TwinNet outperforms both baselines, achieving a MAPE of 7.1%.

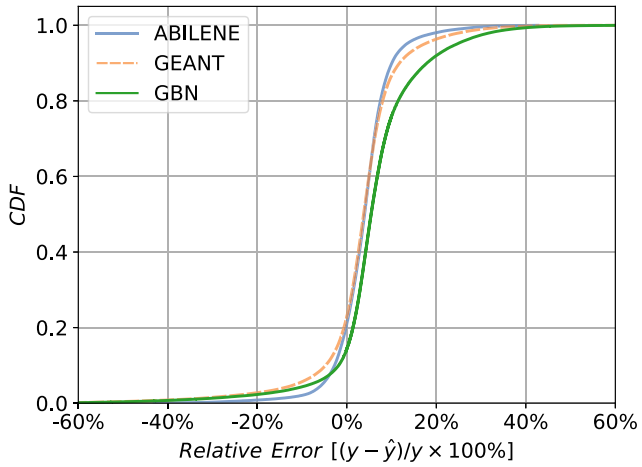


Fig. 8. CDF of the relative error using real traffic.

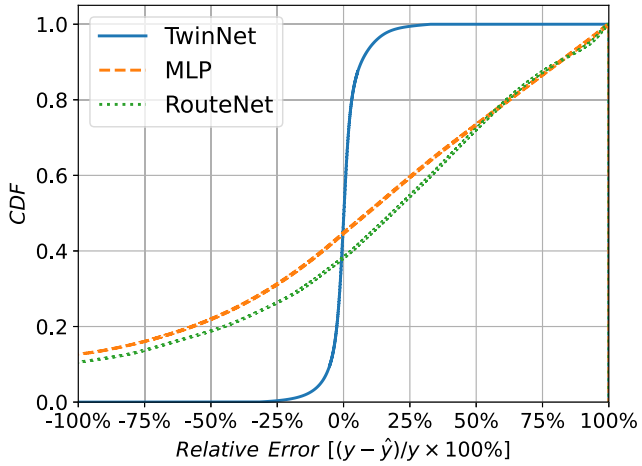


Fig. 9. CDF of relative error over 106 unseen real-world topologies with realistic traffic.

7.6. Speed

An important application of TwinNet is its integration with an optimization algorithm (see Section 2). This requires operating in short timescales to quickly adapt to the network conditions (e.g., traffic changes); especially in novel communication paradigms with stringent QoS requirements (e.g., Tactile Internet [6,7], ultra-low deterministic latency for AR/VR, holographic telepresence [8,9]).

In TwinNet, the inference cost is proportional to the number of nodes, queues, and links in the network, which offers better scalability properties than packet-level simulators. In our experiments, TwinNet obtained average execution times of 42 ms (ms) for the smallest networks in the Internet Topology Zoo (10–15 nodes) and 145 ms for the larger ones (85–95 nodes). In contrast, the average execution time with a packet-level simulator (OMNet++ [37]) was 71 s for the smallest networks (10–15 nodes) and over 1 h (1 h 10mins, on average) for the larger ones (85–95 nodes). These figures represent the cost per sample (e.g., input traffic, routing, and queuing policy). However, they should be multiplied by a large number of combinations (e.g., routing, queuing policies) typically explored by a network optimizer before producing a solution that meets the optimization goals (see Section 8).

The scalability limitations of packet-based simulators are explained by the fact that they have to simulate every single packet transmitted over the network (e.g., several million packets per second for a single 10Gbps link). In contrast, the cost of TwinNet is independent of the number of packets.[1] These experiments were made using a single-thread process in a machine with a CPU of 4 GHz. However, the execution times of TwinNet could be dramatically reduced using GPU/TPU hardware acceleration [59].

8. SLA-driven optimization use cases

In this section, we present several relevant use cases that illustrate the potential of TwinNet for SLA-driven optimization. In all of them, TwinNet is paired with an optimization algorithm in order to produce routing and queuing configurations that meet a set of SLAs. Finally, we evaluate the accuracy of TwinNet using data from a real testbed.

8.1. Methodology

We combine TwinNet with an optimization algorithm based on Direct Search. This algorithm uses a custom search heuristic based on common network metrics (link utilization, traffic, path length, etc.), which guides the exploration within the high-dimensional space of solutions. Note that more sophisticated optimization algorithms can be paired with TwinNet. However, we leave this out of the scope of this paper, as the goal of this section is to showcase how TwinNet can be effectively used for SLA-aware optimization.

In all the experimental setups, we generate traffic flows with two different SLA levels (ToS0=Top priority, ToS1=high priority), and some background traffic labeled as *Best effort*. The goal for the optimizer is to find a configuration that fulfills the predefined SLAs for ToS0 and ToS1 while minimizing the mean delay for the best-effort traffic. In all the experiments, we use the TwinNet model previously trained in NSFNET and GEANT (Section 6.2) and perform optimization over scenarios not seen during training (in the GBN topology).

As a benchmark, we also show the performance of a shortest-path policy using FIFO scheduling on all network devices, as well as the configuration that results from running an optimizer that integrates a fluid model instead of TwinNet.

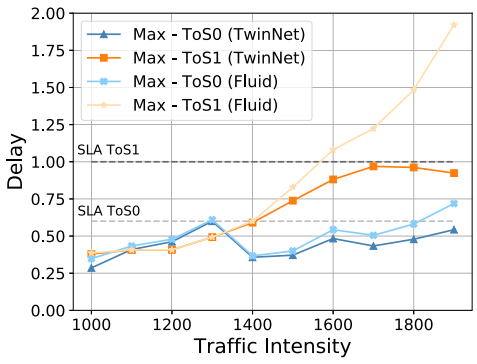
8.2. Routing

In this use case, the goal is to optimize the mean delay of the best-effort traffic while satisfying the set of SLAs for each ToS by only modifying the (src-dst) routing scheme. We evaluate the performance across various traffic intensities ranging from 1000 (middle traffic load) to 1900 (high network congestion).

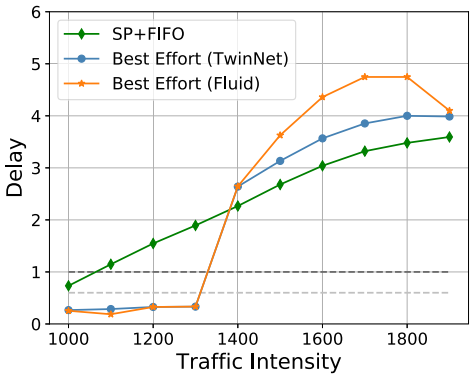
We compare the results obtained using TwinNet with those achieved using a traditional fluid model. This latter case represents a baseline of state-of-the-art optimization tools that rely on fluid models, such as DEFO [14]. However, note that this reference benchmark is not exactly the same as DEFO, as we do not consider middle-point routing or ECMP. Our focus is to make a direct comparison of both network models, under the same conditions. Hence, we use the same optimization algorithm. In both cases, the exploration is guided by the delay estimates of the network model: TwinNet or fluid models. Additionally, we compare the results with the performance of a traditional Shortest Path (SP) policy.

Figs. 10(a) and 10(b) show the results of the optimization. Fig. 10(a) shows how the TwinNet-based optimizer finds a solution that fulfills the SLA requirements for both ToS (top and high priorities). Fig. 10(b) shows that the optimizer paired with TwinNet achieves remarkable performance when minimizing the mean delay of the best-effort traffic, outperforming both the fluid and the SP for low traffic intensities. For high traffic load, TwinNet manages to fulfill the SLAs for both ToS

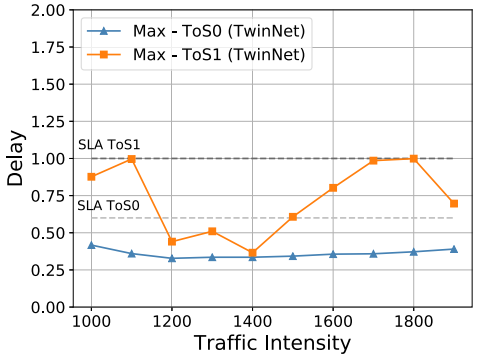
[1] Note that in our experiments we only use a packet simulator during the training phase. After training, TwinNet can produce accurate estimates in the scale of milliseconds for networks unseen in training (see Section 7.4)



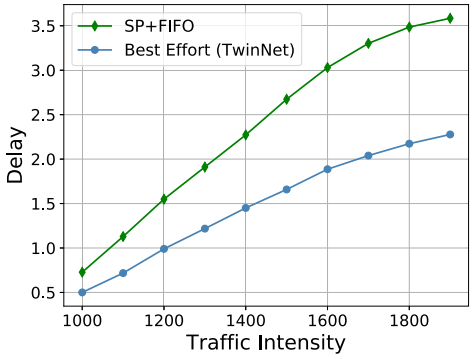
(a) SLA flows - Routing (max. flow mean delay)



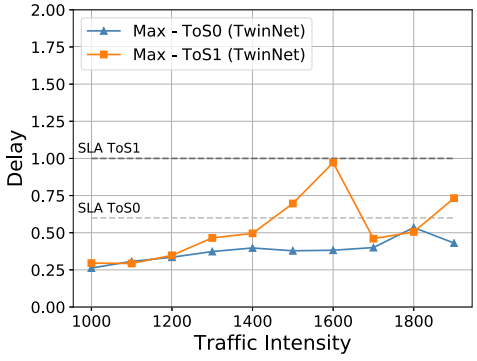
(b) Best Effort - Routing (avg. flow mean delay)



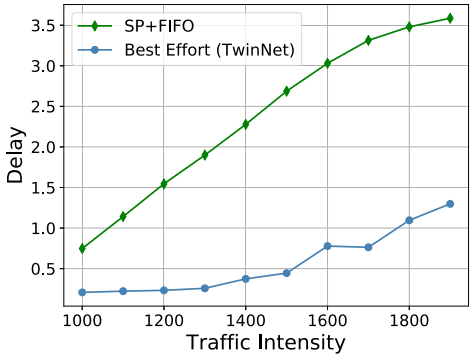
(c) SLA flows - Scheduling (max. flow mean delay)



(d) Best Effort - Scheduling (avg. flow mean delay)



(e) SLA flows - Routing & Scheduling (max. flow delay)



(f) Best Effort - Routing & Scheduling (avg. flow delay)

Fig. 10. SLA-driven optimization with routing (top), scheduling (middle) and both routing & scheduling (bottom)

(Fig. 10(a)), while this has an impact on the mean delay of best-effort traffic (Fig. 10(b)).

Regarding the fluid model, we can observe that the optimizer could not find solutions that satisfy the SLAs for medium to high traffic intensities. The reason for this is that despite the fluid model estimates that the delay experienced by flows is within the SLA terms, the packet-level simulator shows that such estimates are not accurate – as shown earlier in Section 3 – and exceed the SLA thresholds. This is because the fluid model ignores queuing delay. Thus, in highly congested scenarios, where queuing delay becomes more significant, the fluid model-based optimizer leads to SLA violations.

8.3. Scheduling

This use case also attempts to minimize the mean delay on best-effort traffic, while satisfying the SLAs assigned to each ToS. However, in this section, we aim to evaluate the potential of optimizing the queuing configuration using TwinNet. The main difference with the previous experiments is that now the optimizer only explores different queuing configurations, while routing is fixed to a standard shortest path scheme. Then, we leverage the flexibility of TwinNet to make delay estimates under different queue scheduling policies and parameters. Note that in this case, we cannot compare the results with a fluid model

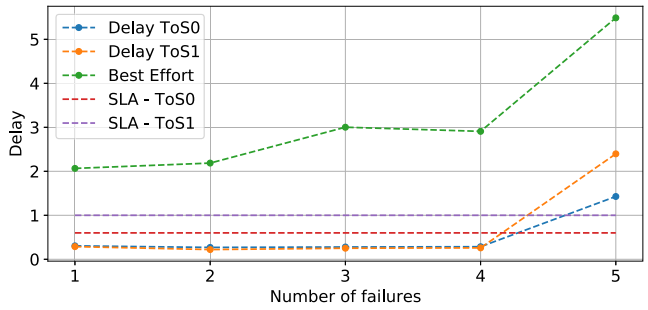


Fig. 11. SLA-driven optimization with link failures.

given that it does not have support for queuing policies beyond FIFO. Because of this, optimizers based on fluid models are typically limited to exploring different routing configurations.

Figs. 10(c) and 10(d) show that the optimizer is able to fulfill all SLAs, while also minimizing the delay of best-effort traffic. More importantly, if we compare these results with the previous ones, we can observe that by modifying the queuing configuration we achieve better results than modifying the routing. These results illustrate the remarkable impact of the queue scheduling configuration in the overall network performance, particularly in the presence of different ToS.

8.4. Routing and scheduling

Based on the previous results, in this section , we aim to evaluate the optimization potential when optimizing both the routing and queuing configuration. The objective is the same as before, minimizing the mean delay on best-effort traffic while satisfying the SLAs.

Figs. 10(e) and 10(f) show the evaluation results. As we can observe the improvement is remarkable compared to the previous results. The TwinNet-based optimizer satisfies all SLAs while pushing the mean delay of best-effort traffic even lower. For instance, in the scenario with the highest traffic load, it achieves a reduction on the mean delay of ≈60% with respect to the SP+FIFO policy.

8.5. Robustness against link failures

In this use case, we evaluate if our model is able to generalize in the presence of link failures. When a certain link fails, we need to find a new routing and queuing configuration to avoid such link. As the number of link failures increases, fewer paths are available and the network becomes more saturated.

We run our TwinNet-based optimizer in scenarios with a number of random link failures. The initial network scenario is the same as in the previous experiments (i.e., routing & scheduling optimization), considering the highest traffic intensity ($TI = 1900$).

Fig. 11 shows the optimized mean delay with respect to the number of link failures (n). Each point in the plot corresponds to the optimal delay obtained over 10 experiments with n random link failures. We observe that the mean delay increases gradually on best-effort traffic as there are more link failures and the network becomes increasingly congested. Nevertheless, the optimizer is able to meet all the SLAs even with up to 4 link failures.

8.6. What-if: Budget-constrained network upgrade

In this last use case, we show how our Digital Twin can be used to reason about the network and provide recommendations. Particularly, we use it to answer the following question: *What is the optimal link upgrade in the network?* The question is put in the context of a network administrator that has a static and well-known traffic matrix and that is willing to upgrade the capacity of the network by adding one link that minimizes the mean traffic delay.

Table 4
Optimal link placement with various TMs.

Traffic matrix	Optimal new link placement	Previous delay	Delay with new link	Delay reduction
TM_1	(6,16)	0.446	0.259	41.9%
TM_2	(8,13)	0.508	0.303	40.3%
TM_3	(7,15)	0.409	0.239	41.5%
TM_4	(6,14)	0.499	0.253	49.3%
TM_5	(7,15)	0.551	0.322	41.5%
TM_6	(6,16)	0.458	0.209	54.3%
TM_7	(5,12)	0.419	0.251	40.1%
TM_8	(6,14)	0.590	0.312	47.1%

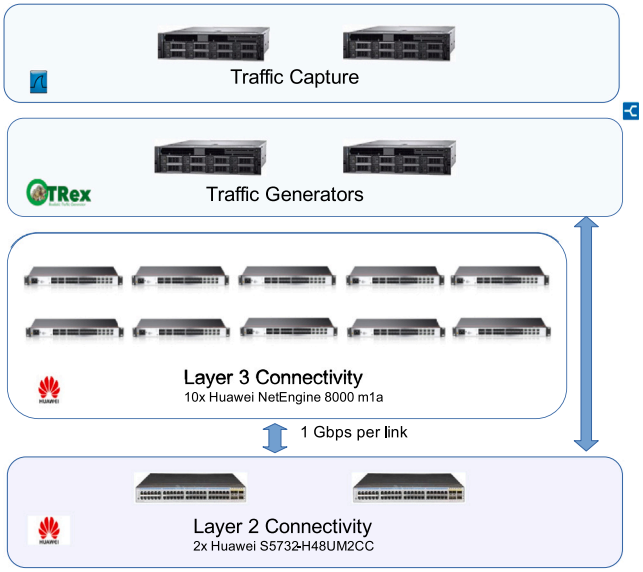


Fig. 12. Schematic representation of the network testbed.

To answer this question we use our TwinNet-based optimizer, constrained to adding only one link. Table 4 shows the results for scenarios with 8 different random traffic matrices of traffic intensity TI=1500. We can see that the optimizer achieves a considerable delay reduction (40.1%–54.3%) by properly selecting the best link placement.

8.7. Evaluating TwinNet in a testbed

In previous sections, we explored the use of TwinNet in several relevant use cases. This section evaluates the performance of TwinNet in a network scenario using real hardware and packet traces similar to the ones used in Section 7.5.

To generate the data used for training and testing the model, we use a physical network testbed, which is depicted in Fig. 12. This testbed comprises the following hardware: (*i*) 8 Routers Huawei NetEngine 8000 M1 A (4*10GE+12*GE) (*ii*) 2 Switches Huawei S5732-H48UM 2CC 5G Bundle (48*100M/1G / 2.5G/5G Ethernet ports), and (*iii*) 4 servers, two of them used for traffic generation, and the the other two used for traffic capture and analysis. We generate traffic using the TRex traffic generator [60], and we capture traffic with the PF_RING software [61].

We generate 6,000 samples with realistic topologies (with a maximum of 8 nodes), a routing and a queueing configuration, and a traffic matrix. From these 6,000 samples, 4,000 are randomly selected for training the model, and the remaining 2,000 samples are used for testing.

Table 5 shows the results obtained when evaluating TwinNet over the 2,000 test samples unseen during the training phase. As it can

Table 5
Accuracy evaluation with data from a real testbed.

	TESTBED			
	MAPE	MSE	MAE	R^2
TwinNet	0.063	2.30×10^{-9}	1.45×10^{-5}	0.915

be seen, TwinNet is able to produce accurate delay estimates in these scenarios from a real testbed. Particularly, the model obtains a MAPE of 6.3% with respect to the real values measured in the testbed. This error is in line with the accuracy levels observed previously in Section 7 with simulated data.

9. Related work

In the context of network optimization, a fundamental goal of network models is providing a cost function that is later used for optimization. Over the years, many attempts have been made to obtain accurate cost functions. This includes fluid models, analytical models (e.g., queuing theory, network calculus), and discrete-event network simulators (e.g., ns-3, OMNet++).

Among all the existing techniques, queuing theory is arguably the most popular [12]. Queuing theory remains an open problem for realistic multi-point to multi-point queuing networks, and holds hard assumptions on traffic models (e.g., packet inter-arrival times). This makes it impractical in several real-world operations. Alternatively, fluid models are efficient and popular for traffic engineering problems. However, they make important simplifications and, as shown in Section 3, produce inaccurate estimates in scenarios where queuing delay is significant. Likewise, network calculus operates over the worst-case scenario of networks [62]. However, this type of scenario is rarely observed in operational environments.

The use of Deep Learning for network modeling has recently attracted a lot of interest from the networking community [26]. The main advantage of this approach is that it is data-driven and, as such, it can potentially model the complexity of real networks. Existing proposals mainly use traditional fully-connected neural networks (e.g., [25,63]). However, the main limitation of these solutions is that they do not generalize to other network topologies and configurations (e.g., routing). In this context, more recent works propose more elaborated NN models like Variational Auto-encoders [18], Convolutional NNs [64], or Graph NNs [65–67]. Nevertheless, these models have fundamentally different goals compared with TwinNet and do not consider the core components of real networks. For instance, they are not able to predict the effect of multi-hop queuing policies along arbitrary paths.

Another recent work [68] proposes to combine GNN with Deep Reinforcement Learning to perform automatic and efficient network optimization. However, they use a basic GNN architecture that considers a simplified model of the network, including only support for link-level features (e.g., capacity, utilization), and is particularly designed to optimize routing and channel allocation in an Optical Transport Network scenario, i.e. a circuit-switched network without queues.

Finally, there is a growing body of work that leverages GNNs in different networking scenarios, such as network planning [69], that exploits the capabilities of GNNs to encode the network topology, and uses Reinforcement Learning to prune the search space of Integer Linear Programming problems. Another application is flow selection: a GNN selects the most critical flow, and it is then rerouted using Linear Programming techniques [70]. Other applications include distributed Traffic Engineering optimization [71] (using a GNN+MARL solution), or load balancing in multi-region networks [72], which leverages a GNN coupled with a collaborative approach. However, we must remark that all of these works take advantage of GNNs to encode the state of the network at a certain moment, but do not directly produce delay estimates. In addition, the scenarios described in these works do not consider the impact of the different scheduling policies.

10. Conclusions

Network models are a central component for network control and optimization, as they enable the evaluation of network performance under alternative configurations and what-if scenarios. Reproducing the behavior of real networks involves a series of complex non-linear relationships among multiple components that define the network state (e.g., topology, routing, queuing configuration, traffic).

In this paper, we presented TwinNet, a GNN-based Digital Twin that implements a custom message-passing NN architecture particularly designed to model these complex relationships. For this purpose, TwinNet explicitly defines network elements and their relations in its internal neural network architecture and learns how to reason on this graph-structured information. We applied TwinNet to estimate per-path delays in networks, covering a wide variety of topologies, configurations (routing and queuing), and traffic load levels. Our evaluation shows that TwinNet is able to generalize accurately to samples of 106 real-world networks (from Internet Topology Zoo) unseen in advance, after being only trained on samples of two different networks (NSFNET and GEANT). Also, our evaluation shows how TwinNet is capable of understanding the complex relationships in a real testbed scenario. This reflects the capability of this model to abstract deep insights from network scenarios seen during training and then apply this knowledge effectively to new network topologies, including different traffic matrices, routing, and queuing configurations.

Finally, we presented several relevant use cases that illustrate the potential of the proposed model for SLA-driven optimization. To this end, we paired TwinNet with an optimization algorithm to produce routing and queuing configurations that meet a set of SLAs in a specific network scenario. We also used it to test the robustness of the network against link failures and to find the optimal link placement in a network planning use case.

CRediT authorship contribution statement

Miquel Ferriol-Galmés: Writing – original draft, Methodology, Conceptualization, Investigation. **José Suárez-Varela:** Writing – review & editing, Resources. **Jordi Paillissé:** Writing – review & editing. **Xiang Shi:** Methodology, Resources. **Shihan Xiao:** Methodology, Resources. **Xiangle Cheng:** Methodology, Supervision. **Pere Barlet-Ros:** Writing – review & editing, Supervision, Conceptualization. **Albert Cabellos-Aparicio:** Writing – original draft, Supervision, Conceptualization.

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.comnet.2022.109329. Albert Cabellos-Aparicio reports financial support was provided by Catalan Institution for Research and Advanced Studies, Spain. Albert Cabellos-Aparicio reports financial support was provided by Spain Ministry of Science and Innovation. Albert Cabellos-Aparicio reports a relationship with Catalan Institution for Research and Advanced Studies, Spain that includes: funding grants. Albert Cabellos-Aparicio reports a relationship with Spain Ministry of Science and Innovation that includes: funding grants.

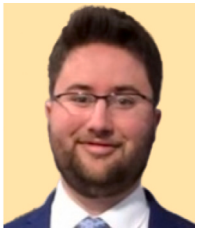
Data availability

All the code as well as all the data used in this submission is publicly available. A link can be found in the Manuscript.

References

[1] Akyildiz, et al., A roadmap for traffic engineering in SDN-OpenFlow networks, Comput. Netw. 71 (2014) 1–30.
[2] F.S. Yiannis Yiakoumis, Network tokens, (draft-yiakoumis-network-tokens-01) 2020, Work in Progress.
[3] Z. Yang, et al., Software-defined wide area network (SD-WAN): Architecture, advances and opportunities, in: International Conference on Computer Communication and Networks, ICCCN, 2019, pp. 1–9.
[4] H. Zhang, et al., Network slicing based 5G and future mobile networks: mobility, resource management, and challenges, IEEE Commun. Mag. 55 (8) (2017) 138–145.
[5] X. Li, D. Li, et al., A review of industrial wireless networks in the context of industry 4.0, Wirel. Netw. 23 (1) (2017) 23–41.
[6] M. Simsek, et al., 5G-enabled tactile internet, IEEE J. Sel. Areas Commun. 34 (3) (2016) 460–473.
[7] R. Gupta, et al., Tactile internet and its applications in 5G era: A comprehensive review, Int. J. Commun. Syst. 32 (14) (2019) e3981.
[8] Z. Chen, et al., NEW IP framework and protocol for future applications, in: IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1–5.
[9] R. Li, Towards a new internet for the year 2030 and beyond, in: Annual ITU IMT-2020/5G Workshop Demo Day, 2018, pp. 1–21.
[10] F. Ciucu, J. Schmitt, Perspectives on network calculus: no free lunch, but still good value, in: ACM SIGCOMM, 2012, pp. 311–322.
[11] J.W. Guck, V. Bemten, et al., Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation, IEEE Commun. Surv. Tutor. 20 (1) (2017) 388–415.
[12] G. Giambene, Queuing Theory and Telecommunications, vol. 585, Springer, 2014.
[13] Z. Xu, et al., Experience-driven networking: A deep reinforcement learning based approach, in: IEEE INFOCOM, 2018, pp. 1871–1879.
[14] R. Hartert, et al., A declarative and expressive approach to control forwarding paths in carrier-grade networks, ACM SIGCOMM Comput. Commun. Rev. 45 (4) (2015) 15–28.
[15] L. Guo, I. Matta, Search space reduction in QoS routing, Comput. Netw. 41 (1) (2003) 73–88.
[16] M. Jadin, F. Aubry, P. Schaus, O. Bonaventure, CG4SR: Near optimal traffic engineering for segment routing with column generation, in: IEEE INFOCOM, 2019, pp. 1333–1341.
[17] R. Bhatia, et al., Optimized network traffic engineering using segment routing, in: IEEE INFOCOM, 2015, pp. 657–665.
[18] S. Xiao, et al., Deep-q: Traffic-driven qos inference using deep generative network, in: ACM SIGCOMM Workshop on Network Meets AI & ML, 2018, pp. 67–73.
[19] V. Mnih, et al., Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.
[20] Y. LeCun, et al., Deep learning, Nature 521 (7553) (2015) 436–444.
[21] X.D. Cheng Zhou, Concepts of digital twin network, (draft-zhou-nmrg-digitaltwin-network-concepts-00) 2020, Work in Progress.
[22] H. Laaki, et al., Prototyping a digital twin for real time remote control over mobile networks: Application of remote surgery, IEEE Access 7 (2019) 20325–20336.
[23] H.X. Nguyen, R. Trestian, D. To, M. Tatipamula, Digital twin for 5G and beyond, IEEE Commun. Mag. 59 (2) (2021) 10–15.
[24] J. Autiosalo, Platform for industrial internet and digital twin focused education, research, and innovation: Ilmatar the overhead crane, in: IEEE World Forum on Internet of Things (WF-IoT), 2018, pp. 241–244.
[25] A. Mestres, et al., Understanding the modeling of computer network delays using neural networks, in: ACM SIGCOMM BigDAMA Workshop, 2018, pp. 46–52.
[26] M. Wang, et al., Machine learning for networking: Workflow, advances and opportunities, IEEE Netw 32 (2) (2017) 92–99.
[27] F. Scarselli, M. Gori, et al., The graph neural network model, IEEE Trans. Neural Netw. 20 (1) (2008) 61–80.
[28] O. Bousquet, A. Elisseeff, Stability and generalization, J. Mach. Learn. Res. 2 (2002) 499–526.
[29] S. Knight, H. Nguyen, et al., The internet topology zoo, IEEE J. Sel. Areas Commun. (ISSN: 0733-8716) 29 (9) (2011) 1765–1775, http://dx.doi.org/10.1109/JSAC.2011.111002.
[30] F. Krasniqi, J. Elias, J. Leguay, A.E. Redondi, End-to-end delay prediction based on traffic matrix sampling, in: IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2020, pp. 774–779.
[31] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, A. Cabellos-Aparicio, RouteNet: Leveraging graph neural networks for network modeling and optimization in SDN, IEEE J. Sel. Areas Commun. 38 (10) (2020) 2260–2270.
[32] C. Kim, et al., In-band network telemetry via programmable dataplanes, in: ACM SIGCOMM, Posters and Demos, 2015, pp. –.
[33] M. Yu, L. Jose, R. Miao, Software defined traffic measurement with OpenSketch, in: Symposium on Networked Systems Design and Implementation, NSDI, 2013, pp. 29–42.
[34] R.B. Cooper, Queueing theory, in: Proceedings of the ACM'81 Conference, 1981, pp. 119–122.
[35] K. Weiss, T.M. Khoshgoftaar, D. Wang, A survey of transfer learning, J. Big Data 3 (1) (2016) 9.
[36] S. Gay, P. Schaus, S. Vissicchio, Repetita: Repeatable experiments for performance evaluation of traffic-engineering algorithms, 2017, arXiv preprint arXiv:1710.08665.
[37] A. Varga, Discrete event simulation system, in: European Simulation Multiconference, ESM, 2001, pp. 1–7.
[38] X. Hei, J. Zhang, et al., Wavelength converter placement in least-load-routing-based optical networks using genetic algorithms, J. Opt. Netw. 3 (5) (2004) 363–378.
[39] F. Barreto, et al., Fast emergency paths schema to overcome transient link failures in ospf routing, 2012, arXiv preprint arXiv:1204.2465.
[40] J. Pedro, J. Santos, J. Pires, Performance evaluation of integrated OTN/DWDM networks with single-stage multiplexing of optical channel data units, in: International Conference on Transparent Optical Networks, 2011, pp. 1–4.
[41] P.W. Battaglia, et al., Relational inductive biases, deep learning, and graph networks, 2018, arXiv preprint arXiv:1806.01261.
[42] J. Gilmer, et al., Neural message passing for quantum chemistry, 2017, arXiv preprint arXiv:1704.01212.
[43] P. Battaglia, et al., Interaction networks for learning about objects, relations and physics, in: Advances in Neural Information Processing Systems, 2016, pp. 4502–4510.
[44] P. Gainza, et al., Deciphering interaction fingerprints from protein molecular surfaces using geometric deep learning, Nature Methods 17 (2) (2020) 184–192.
[45] W. Fan, et al., Graph neural networks for social recommendation, in: The ACM World Wide Web Conference, WWW, 2019, pp. 417–426.
[46] D. Raposo, et al., Discovering objects and their relations from entangled scene representations, 2017, arXiv preprint arXiv:1702.05068.
[47] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks? 2018, arXiv preprint arXiv:1810.00826.
[48] D.E. Rumelhart, et al., Learning representations by back-propagating errors, Nature 323 (6088) (1986) 533–536.
[49] J. Zhou, et al., Graph neural networks: A review of methods and applications, 2018, arXiv preprint arXiv:1812.08434.
[50] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014, arXiv preprint arXiv:1412.3555.
[51] G. Klambauer, T. Unterthiner, A. Mayr, S. Hochreiter, Self-normalizing neural networks, in: Advances in Neural Information Processing Systems, NIPS, 2017, pp. –.
[52] M. Ferriol-Galmés, Building a digital twin for network optimization using graph neural networks, 2021, GitHub Repository https://github.com/MiquelFerriol/Papers/wiki/TwinNet.
[53] S. Floyd, V. Paxson, Difficulties in simulating the internet, IEEE/ACm Trans. Netw. 9 (4) (2001) 392–403.
[54] M.E. Newman, S.H. Strogatz, D.J. Watts, Random graphs with arbitrary degree distributions and their applications, Phys. Rev. E 64 (2) (2001) 026118.
[55] A. Botchkarev, Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology, 2018, arXiv preprint arXiv:1809.03006.
[56] S. Orlowski, R. Wessäly, M. Pióro, A. Tomaszewski, SNDlib 1.0—Survivable network design library, Netw. Int. J. 55 (3) (2010) 276–286, URL http://sndlib.zib.de.
[57] R. Fontugne, P. Borgnat, P. Abry, K. Fukuda, Mawilab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking, in: ACM CoNEXT '10, Philadelphia, PA, 2010, pp. –.
[58] S. Schnitter, et al., Quality-of-service class specific traffic matrices in IP/MPLS networks, in: ACM Internet Measurement Conference, 2007, pp. 253–258.
[59] M. Gallo, A. Finamore, G. Simon, D. Rossi, Real-time deep learning based traffic analytics, ACM SIGCOMM, Demo Session (2020).
[60] Cisco systems TRex: Cisco's realistic traffic generator, 2022, https://trex-tgn.cisco.com/. (Accessed 16 May 2022).
[61] NTOP PF_RING: High-speed packet capture, filtering and analysis, 2022, https://www.ntop.org/products/packet-capture/pf_ring/. (Accessed 06 May 2022).
[62] J.-Y. Le Boudec, P. Thiran, Network Calculus: A Theory of Deterministic Queuing Systems for the Internet, vol. 2050, Springer Science & Business Media, 2001.
[63] A. Valadarsky, et al., Learning to route, in: ACM Workshop on Hot Topics in Networks, 2017, pp. 185–191.
[64] X. Chen, et al., Deep-RMSA: A deep-reinforcement-learning routing, modulation and spectrum assignment agent for elastic optical networks, in: 2018 Optical Fiber Communications Conference and Exposition, OFC, 2018, pp. 1–3.
[65] F. Geyer, S. Bondorf, DeepTMA: Predicting effective contention models for network calculus using graph neural networks, in: IEEE INFOCOM, 2019, pp. 1009–1017.
[66] K. Rusek, J. Suárez-Varela, et al., Unveiling the potential of graph neural networks for network modeling and optimization in SDN, in: ACM Symposium on SDN Research, 2019, pp. 140–151.
[67] Z. Meng, et al., Interpreting deep learning-based networking systems, in: ACM SIGCOMM, 2020, pp. 154–171.

[68] P. Almasan, J. Suárez-Varela, et al., Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case, 2019, arXiv preprint arXiv:1910.07421.
[69] H. Zhu, V. Gupta, S.S. Ahuja, Y. Tian, Y. Zhang, X. Jin, Network planning with deep reinforcement learning, in: Proceedings of the 2021 ACM SIGCOMM 2021 Conference, 2021, pp. 258–271.
[70] M. Ye, J. Zhang, Z. Guo, H.J. Chao, Date: Disturbance-aware traffic engineering with reinforcement learning in software-defined networks, in: 2021 IEEE/ACM 29th International Symposium on Quality of Service, IWQOS, IEEE, 2021, pp. 1–10.
[71] G. Bernárdez, J. Suárez-Varela, A. López, B. Wu, S. Xiao, X. Cheng, P. Barlet-Ros, A. Cabellos-Aparicio, Is machine learning ready for traffic engineering optimization? in: 2021 IEEE 29th International Conference on Network Protocols (ICNP), IEEE, 2021, pp. 1–11.
[72] M. Ye, J. Zhang, Z. Guo, H.J. Chao, Federated traffic engineering with supervised learning in multi-region networks, in: 2021 IEEE 29th International Conference on Network Protocols, ICNP, IEEE, 2021, pp. 1–12.



Miquel Ferriol-Galmés received his B.Sc. degree in Computer science and his M.Sc. degree in Data Science respectively in 2018 and 2020 from the Universitat Politècnica de Catalunya (UPC). He is currently pursuing a Ph.D. at the Barcelona Neural Networking center (BNN-UPC). His main research interests are in the application of Graph Neural Networks to network modeling and optimization, as well as in the application of blockchain to networks.

José Suárez-Varela is currently a postdoctoral researcher in the Barcelona Neural Networking center (BNN-UPC). He holds a Ph.D. in Computer Science from the Polytechnic University of Catalonia (UPC) in 2020, and a B.Sc and M.Sc. degrees in Telecommunications Engineering from the University of Granada (UGR). He was co-PI of the EU-funded project IGNNITION (H2020 NGI POINTER program) and, during 2020, he was a member of the Management Board of the ITU AI/ML in 5G Challenge. During 2019, he was a visiting researcher at the University of Siena (Italy), under the supervision of Prof. Franco Scarselli. His main research interests are in the field of Machine Learning for network control and management, traffic measurement and analysis, and cybersecurity.

Jordi Paillissé received a degree in Telecommunications Engineering (2013) and a PhD in Computer Architecture (2021) from the Technical University of Catalonia — BarcelonaTech. He has been a visiting student at École Polytechnique Fédérale de Lausanne (Switzerland), and a visiting researcher at Cisco Systems (USA). He is currently a postdoctoral researcher in the Barcelona Neural Networking Center, working at the intersection of computer networks and machine learning. His main research interests are network modeling, future Internet architectures, Software-Defined Networking and blockchain applications for the Internet.

Xiang Shi received her Bachelor's degree from the Minzu University of China, in 2014, and her PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences in 2020. Currently she works in the Network Technology Laboratory at Huawei Technologies.

Shihan Xiao received the B.Eng. degree in Electronic and Information Engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2012, and his Ph.D. degree in the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2017. He is currently a technical expert of Network AI at Huawei Technologies. His research interests are in the areas of networking and machine learning.

Xiangle Cheng is currently a research fellow in Huawei 2012 Lab. He received the M.Sc. degree in communication and information system from Southwest Jiaotong University, China in 2015, and Ph.D. degree in Computer Science from the University of Exeter, UK in 2020. His research interests include 5G SDN/NFV, Network AI, Stochastic & Neural Combinatorial Optimization, Intelligent Wireless Networks and Mobile Computing, and Dynamic System Modeling and Performance Optimization.

Pere Barlet-Ros is an associate professor at Universitat Politècnica de Catalunya (UPC) and scientific director of the Barcelona Neural Networking center (BNN-UPC). From 2013 to 2018, he was co-founder and chairman of the machine learning startup Talaia Networks, which was acquired by Auvik Networks in 2018. He was also a visiting researcher at Endace (New Zealand), Intel Research Cambridge (UK), and Intel Labs Berkeley (USA). His research has been integrated with several open-source and commercial products, including Talaia Polygraph, Auvik TrafficInsights, Intel CoMo, and SMARTxAC. In 2015, he was awarded as the best entrepreneur of the UPC School of Computer Science (FIB). He received a Ph.D. degree in Computer Science from UPC in 2008.

Albert Cabellos-Aparicio is an associate professor at Universitat Politècnica de Catalunya. Albert has led several industrial projects, including collaborations with Cisco, Samsung, Intel, and Huawei. In addition, he has participated as a PI in several publicly funded projects, including FET-OPEN (H2020) and the Spanish Funding Agency (CICYT). He has also founded the Open Overlay Router open-source initiative (http://openoverlayrouter.org) along with Cisco. He has been a visiting researcher at Cisco Systems and Agilent Technologies and a visiting professor at the Royal Institute of Technology (KTH), the Massachusetts Institute of Technology (MIT), and UC Berkeley. Since 2015 he is the Vice-Dean for International and Institutional Relations at FIB-UPC. He has co-authored more than 50 journals, over 100 conference papers and participated in around 10 IETF documents. Albert was recently awarded by ICREA Academia.