Performance Evaluation 130 (2019) 1–16

Contents lists available at ScienceDirect

Performance Evaluation

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

DeepComNet: Performance evaluation of network topologies using graph-based deep learning

Fabien Geyer

Technical University of Munich, Boltzmannstr. 3, 85748 Garching b. München, Germany

ARTICLE INFO

Article history:
Available online 28 December 2018

Keywords:
Network performance evaluation
Graph neural network
Deep learning

ABSTRACT

Modeling the performance of network protocols and communication networks generally relies on expert knowledge and understanding of the different elements of a network, their configuration, and the overall architecture and topology. Machine learning is often proposed as a tool to help modeling such complex systems. One drawback of this method is that high-level features are generally used – which require full understanding of the network protocols to be chosen, correctly engineered, and measured – and the approaches are generally limited to a given network topology.

In this article, we present *DeepComNet*, an approach to address the challenges of working with machine learning by using lower-level features, namely only a description of the network architecture. Our main contribution is a method for applying deep learning on network topologies via the use of Graph Gated Neural Networks, a specialized recurrent neural network for graphs. Our approach enables us to make performance predictions based only on a graph-based representation of network topologies. To evaluate the potential of DeepComNet, we apply our approach to the tasks of predicting the throughput of TCP flows and the end-to-end latencies of UDP flows. In both cases, the same base model is used. Numerical results show that our approach is able to learn and predict performance properties of TCP and UDP flows with a median absolute relative error smaller than 1%, outperforming related methods from the literature by one order of magnitude.

1. Introduction

Understanding network performance is an important task for architecture design and Quality-of-Service in an increasing number of applications. Traffic engineering aims at bringing an answer to this need in order to provide better services, avoid congestion, and optimize network topologies to support an increasing number of applications. Models for network and traffic are an important tool in order to predict how a given network architecture will behave. Different techniques have been proposed, such as mathematical modeling, simulations or measurements in real networks. While these techniques can achieve accurate results, they often require precise measurements of key performance indicators such as round-trip time or loss probability in order to be applied and generate realistic performance predictions. However, limited access to instrumentation of real networks makes this measurement acquisition usually difficult.

An approach to tackle the challenges of performance modeling is a more data-driven way, where measurements are used in parallel with machine learning to produce realistic performance predictions. For instance, Tian and Liu [1] applied the SVR-based (Support Vector Regression) TCP bandwidth prediction application from [2] to improve Quality-of-Service of media streaming over HTTP. Tariq et al. [3] recently proposed WISE, a framework for evaluating architecture changes

E-mail address: fgeyer@net.in.tum.de.

https://doi.org/10.1016/j.peva.2018.12.003
0166-5316/

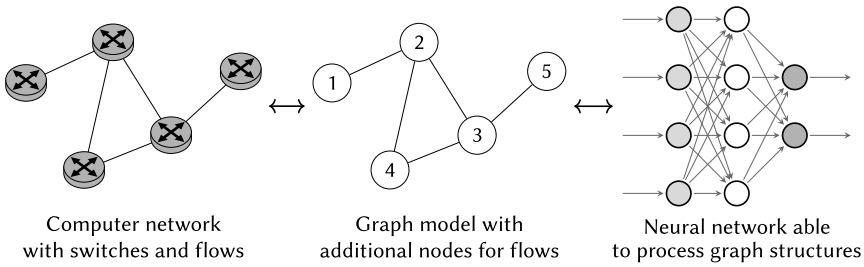


Fig. 1. Illustration of the two different steps of our approach *DeepComNet*.

in communication networks using Causal Bayesian Networks (CBNs). While these techniques and applications have been proven successful, high-level features about the studied network protocols are required, and the trained models are often limited to a given network topology.

Our main contribution in this work is *DeepComNet*, an approach combining formal modeling of network topologies and flows traversing them as graphs with deep learning methods tailored for training and inference on graph structures. This approach is illustrated in Fig. 1 with its two main steps: we first model computer networks as graph structures, and then process those graphs using neural networks. We propose to use Gated Graph Neural Networks (GGNNs) [4] as a basis for our deep learning architecture, a recent extension of Graph Neural Networks (GNNs) [5,6]. By directly training the underlying machine learning algorithm on the network structure, we avoid the task of engineering high-level protocol-specific input features such as round-trip time or drop probability, which usually require expert knowledge on the network protocol which is modeled. We demonstrate in this paper this modeling as graph is independent of a given network architecture or network protocol, providing a wide range of applicability compared to other methods.

As concrete applications of our approach, we address the two following tasks. First, we are interested in the performance evaluation of TCP flows, with the goal of predicting the average throughput of each flow in a given topology. Secondly, we look at the performance evaluation of UDP flows and the prediction of end-to-end latencies (average and quantile value). We use the same model for both tasks, highlighting the general applicability of our approach and potential for application to other use-cases for performance evaluation of network topologies and protocols.

We show through a numerical evaluation that our approach is able to predict the performance of TCP and UDP flows by only using this low-level graph-based representation of the network topology and its flows. Overall, the predictions of bandwidths and end-to-end latencies have a median absolute relative error smaller than 1%. The results for the TCP dataset are also compared against two approaches using high-level input features (round-trip time and loss probability), namely Support Vector Regression (SVR) as used in [1,2] and a standard feed-forward neural network. Compared to both approaches, our method achieves better accuracy, with a median relative error one order of magnitude lower compared to SVR.

We also illustrate the scalability of our approach in terms of execution time by numerically evaluating our implementation of GNN. Our measurements illustrate that our implementation scales quadratically with the number of edges, and linearly with the number of nodes. We show that our approach is able to process graphs with up to 1000 nodes and 150 000 edges with a prediction time below 200 us. Finally we illustrate a method for visualizing the learned model in order to apply it to more general questions such as root-cause analysis or bottleneck identification.

This work is structured as follows. In Section 2, we present similar research studies. We describe in Sections 3 and 4 our modeling approach and the neural network architecture used here, with an introduction to Graph Neural Networks and Gated Graph Neural Networks, followed by the application of those concepts to network topologies and flows. We numerically evaluate our approach in Section 5 with the prediction of performance of TCP and UDP flows, compare against two other machine learning approaches, and evaluate our approach in terms of scalability. Section 6 gives some insights on interpreting the learned weights of the neural network. Finally, Section 7 concludes our work.

2. Related work

As shown in a recent survey by Fadlullah et al. [7], machine learning has been used for computer networks for various applications such as traffic classification, flow prediction, and mobility prediction. It has also been applied in the area of performance evaluation of communication networks. Mirza et al. [2] used Support Vector Regression (SVR) using input features such as transfer duration of files over TCP and active measurements in order to measure queuing latency, loss probability and available bandwidth. Tariq et al. [3] proposed WISE to study application performance using Causal Bayesian Networks (CBNs) in order to answer "what-if" questions. Hours et al. [8] also used CBNs to predict the throughput distribution of TCP flows, using similar features as in [2]. While these works proved to deliver accurate results, they are essentially based on high-level engineered features dependent on the studied protocol and those models are not easily transferable to other protocols. In case of TCP flows, these high level features usually include round-trip time and packet loss – following well known mathematical models such as [9] – which also often require active measurement. By having a simple process for abstracting the network topology as graph without such high-level feature, the approach we propose in this article does not

require such expert knowledge or hand engineered features and is better suited to be used on multiple use-cases or multiple network protocols. Our approach is also able to reason about performance evaluation at network level and not at individual node level.

Various methods have been proposed for applying machine learning to graphs-based structures, either based on a spectral or spatial approach. Spectral approaches [10,11] are usually based on the Graph Laplacian, an analogous method to the Discrete Fourier Transform, which transforms graph signals to a spectral domain. The main limitation of these approaches is that the input graph samples are required to be homogeneous. Concretely this means that only a fixed subset of network types may be analyzed and multiple machine learning models may be required in case multiple network types are required.

Spatial approaches do not require a homogeneous graph structure, meaning that they can be applied to a broader range of problems. Gori et al. proposed in [5,6] the Graph Neural Networks (GNNs) architecture, which propagates hidden representations of nodes to its adjacent nodes until a fixed point is reached. GNNs were applied on different tasks such as object localization, ranking of web pages, document mining, or prediction of graph properties [12]. This neural network architecture was subsequently refined in different works. Li et al. [4] proposed Gated Graph Neural Networks (GGNNs), an extension of GNNs with application of more modern practice of neural networks, namely by using Gated Recurrent Units (GRU) [13]. GNNs were evaluated against basic logical reasoning tasks and program verification in [4] and shown to be as-good-as or better than previous state of the art methods. Related approaches were proposed by Kipf and Welling [14] with Graph Convolutional Networks (GCN) and later extended by Schlichtkrull et al. [15] with Relational Graph Convolutional Networks (R-GCNs). GCN also use hidden node state information propagated across edges of the graph via convolutions, while taking into account the type and direction of an edge. Finally, Battaglia et al. [16] recently introduced the graph networks (GN) framework with the goal of providing a unified formalization of many concepts applied in GNNs and extensions of GNNs.

Those methods have been used in a variety of applications related to the study of data which can be modeled as graphs. Grover and Leskovec [17] applied it to link prediction and classification in social networks, protein interactions and natural language processing. Marcheggiani and Titov [18] studied semantic role labeling in natural language processing. Gilmer et al. [19] used GNNs for the prediction of chemical properties of molecules. Allamanis et al. [20] performed source code analysis by modeling source code and interaction between program variables as graphs, in order to predict variable naming. Selsam et al. [21] modeled the interaction between variables in boolean satisfiability problems (SAT) as graphs, with the goal of predicting their satisfiability. In all those applications, the results of graph-based deep learning were similar to or better than previously existing machine learning approaches.

On the challenge of predicting the performance of TCP flows, analytical models have been proposed since the late 1990s. Mathis et al. [22], and subsequently Padhye et al. [9], modeled the throughput of a single flow using TCP Reno as a function of round-trip time, drop probability and some configuration parameters of TCP. This work was then extended by Cardwell et al. [23] to take into account the slow-start phase of TCP. While these models address the mathematical modeling of a single flow, the interaction between multiple flows on a given topology is of greater interest for the problem addressed in this paper. Firoiu et al. [24] proposed to reuse the results from [9] to analyze complete topologies. Those analytical models give great insights in the performance of TCP, but they usually suffer from poor applicability to real-world use-cases due to newer versions of TCP, simplifications of the mathematical model, or lack of modeling of non-intuitive behavior of TCP such as ACK compression or TCP Incast. Some later works have partially addressed those shortcomings, such as the work from Velho et al. [25] and Geyer et al. [26].

This article is an extension of our previous work [27], which applied GGNNs to the performance evaluation of TCP flows. In this article, we propose more advanced GGNNs architectures, provide an extended numerical evaluation with the performance of both TCP and UDP flows, compare against two other machine learning approaches using high-level features, and provide a discussion regarding scalability.

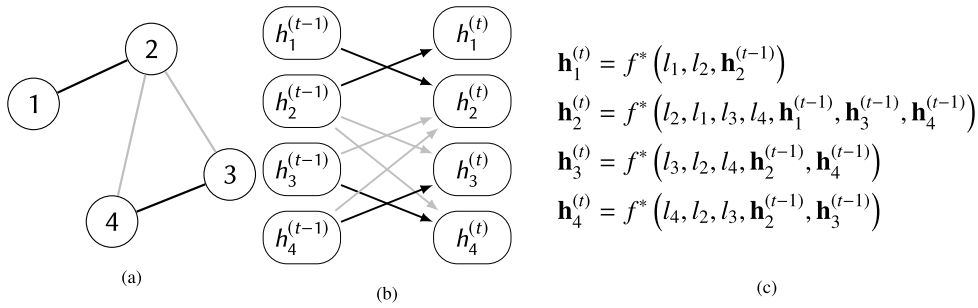
3. Neural networks for graph analysis

The main intuition behind our approach is to represent network topologies as graphs with additional nodes for modeling flows and additional edges for the paths followed by the flows. These graph representations are then used as input for a neural network architecture able to process general graphs. The transformation between a given network topology and its graph representation will be detailed later in Section 4.

In this section, we review the neural network architecture used for training neural networks on graphs, namely Graph Neural Networks (GNNs) [5,6] and one of its recent extension, Gated Graph Neural Networks (GGNNs) [4]. We also introduce notation and concepts that will be used throughout this article. Alternate approaches for applying neural networks to graphs were presented in Section 2.

GNNs and GGNNs are a general neural network architecture able to process graph structures as input. They are an extension of recursive neural networks which work by assigning hidden states to each node in a graph based on the hidden states of adjacent nodes. For the purpose of this work, our description of GNNs and GGNNs is limited to undirected graphs. The concepts presented here can also be applied to directed graph, as presented in the original works on GNNs and GGNNs [4–6].

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. Edges can be represented as pairs of nodes, such that $e = (v, v') \in \mathcal{V} \times \mathcal{V}$. The *hidden representation* for node v is denoted by the vector $\mathbf{h}_v \in \mathbb{R}^{\mathcal{H}}$. Nodes may also have features l_v for each node v, and edges also $l_e = l_{(v,v')}$ for each edge $e = (v, v')$. Let $\textsc{Nbr}(v)$ denote the set of neighboring nodes of v.



$$\mathbf{h}_1^{(t)} = f^*\left(l_1, l_2, \mathbf{h}_2^{(t-1)}\right)$$
$$\mathbf{h}_2^{(t)} = f^*\left(l_2, l_1, l_3, l_4, \mathbf{h}_1^{(t-1)}, \mathbf{h}_3^{(t-1)}, \mathbf{h}_4^{(t-1)}\right)$$
$$\mathbf{h}_3^{(t)} = f^*\left(l_3, l_2, l_4, \mathbf{h}_2^{(t-1)}, \mathbf{h}_4^{(t-1)}\right)$$
$$\mathbf{h}_4^{(t)} = f^*\left(l_4, l_2, l_3, \mathbf{h}_2^{(t-1)}, \mathbf{h}_3^{(t-1)}\right)$$

Fig. 2. (a) Example graph. Edge colors denote edge types. (b) One step of the propagation of hidden node representation. (c) Application of Eq. (1) to the graph.

3.1. Graph neural networks

In Graph Neural Networks (GNNs), each hidden representation $\mathbf{h}_v$ of a node v is based on the hidden state of its neighboring nodes $\text{NBR}(v)$. The following propagation model is used for expressing this relationship:

$$\mathbf{h}_v^{(t)} = f^*\left(l_v, l_{\text{NBR}(v)}, \mathbf{h}_{\text{NBR}(v)}^{(t-1)}\right) \tag{1}$$

An illustrative application of Eq. (1) is given in Fig. 2.

As a concrete implementation, [6] recommends to decompose $f^*(\cdot)$ as the sum of per-edge terms such that:

$$\mathbf{h}_v^{(t)} = \sum_{v' \in \text{NBR}} f\left(l_v, l_{v'}, l_{(v,v')}, \mathbf{h}_{v'}^{(t-1)}\right) \tag{2}$$

with $f(\cdot)$ a linear function of $\mathbf{h}_v$ or a feed-forward neural network. For example, $f(\cdot)$ can be formulated as a linear function:

$$f\left(l_v, l_{v'}, l_{(v,v')}, \mathbf{h}_{v'}^{(t)}\right) = \mathbf{W}^{(l_v, l_{v'}, l_{(v,v')})}\mathbf{h}_{v'}^{(t-1)} + \mathbf{b}^{(l_v, l_{v'}, l_{(v,v')})} \tag{3}$$

with $\mathbf{W}$ and $\mathbf{b}$ learnable weight and bias parameters. The hidden node representations are propagated throughout the graph until a fixed point is reached. As explained in [6], it implies that $f(\cdot)$ has the property that a fixed point for Eq. (2) can be reached.

Once a fixed point $\mathbf{h}_v$ has been reached, a second model is then used to compute the output vector $\mathbf{o_v}$ for each node $v \in \mathcal{V}$ such that:

$$\mathbf{o_v} = g\left(\mathbf{h}_v, l_v\right) \tag{4}$$

Practically, $g(\cdot)$ is implemented using a feed-forward neural network [6]. The neural network architecture is differentiable from end-to-end, so that all parameters can be learned using standard techniques for neural networks based on gradient-based optimization.

Due to the required fixed-point iterations, training of the parameters of $f(\cdot)$ and $g(\cdot)$ of the GNN is done via the Almeida–Pineda algorithm [28,29] which works by running the propagation of the hidden representation to convergence, and then computing gradients based upon the converged solution.

3.2. Gated graph neural networks

Li et al. [4] recently introduced Gated Graph Neural Networks (GGNNs) as an extension of GNNs using more recent neural network techniques, based on Gated Recurrent Units (GRU) [13]. GRUs are special types of neural network blocks with an internal memory. Such blocks are generally used to process time-dependent inputs such as audio or written words, which can be mathematically simplified as:

$$output^{(t)} = function\left(input^{(t)}, output^{(t-1)}\right) \tag{5}$$

In GGNNs, each node aggregates the hidden representations it receives from all adjacent nodes as in Eq. (2), and uses that to update its own hidden representation using a GRU cell. More specifically, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)}\left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \tag{6}$$
$$\mathbf{z}_v^{(t)} = \sigma\left(\mathbf{W}_z a_v^{(t)} + \mathbf{U}_z \mathbf{h}_v^{(t-1)} + \mathbf{b}_z\right) \tag{7}$$
$$\mathbf{r}_v^{(t)} = \sigma\left(\mathbf{W}_r a_v^{(t)} + \mathbf{U}_r \mathbf{h}_v^{(t-1)} + \mathbf{b}_r\right) \tag{8}$$

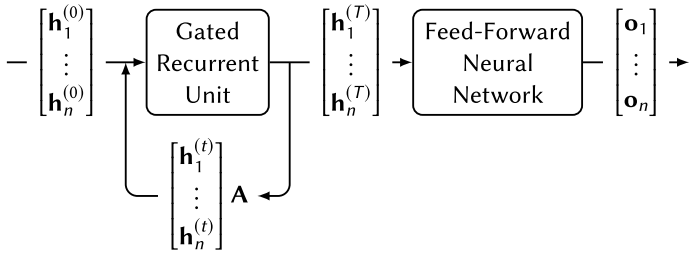


Fig. 3. Representation of a Gated Graph Neural Network.

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}a_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^{(t)} \odot \mathbf{h}_v^{(t-1)}\right) + \mathbf{b}\right) \tag{9}$$

$$\mathbf{h}_v^{(t)} = \left(1 - \mathbf{z}_v^{(t)}\right) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{h}_v^{(t)}} \tag{10}$$

where $\sigma(x) = 1/(1 + e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\{\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}\}$ and $\{\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}\}$ are learnable weights matrices, and $\{\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}\}$ are learnable biases vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix, determining how nodes in the graph $\mathcal{G}$ communicate with each other.

Eq. (6) corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for Graph Neural Networks in Eqs. (1) and (2). Eqs. (7)–(10) correspond to the mathematical formulation of a GRU cell, with Eq. (7) representing the GRU reset gate vector, Eq. (8) the GRU update gate vector, and Eq. (10) the GRU output vector as described in Eq. (5). The initial hidden representation $\mathbf{h}_v^{(0)}$ is based on the node's feature vector l_v, padded with zeros according to the dimensions of the hidden representation.

The output vector $\mathbf{o}_v$ for each node v is computed as in Eq. (4) using a feed-forward neural network. The overall architecture of the GGNN is illustrated and summarized in Fig. 3.

While with GNNs the node propagation loop is performed until a fixed-point is reached, a fixed number of iterations is used in GGNNs. This process enables the use of traditional gradient-based training methods used for feed-forward neural networks such as RMSProp [30]. Since hidden representations are propagated iteratively to node neighbors at each iteration, the number of iterations necessary for achieving good accuracy depends mainly on the path length between relevant nodes in the analyzed graph. In the example in Fig. 2, if the output vector of node 1 depends on the input features of node 4, at least two iterations of the propagation loop are required, since the hidden representation of node 4 is first propagated to node 2 in the first iteration, before reaching node 1 in the second iteration.

While the number of iterations is highly dependent on application, we propose here two simple heuristics for choosing the number of iterations. The first heuristic is to use the diameter d of the studied graph, namely the greatest distance between any pair of nodes, which ensures that each node receives at least once the propagated hidden representation of all other nodes in the graph. The second heuristic is to use $2 \times d$, which ensures that the hidden representations between any pairs of nodes in the studied graph could be propagated in both directions. Numerical evaluations in Section 5.7 illustrate the influence of the number of iteration on the accuracy.

3.3. GGNN-LSTM: Extension of gated graph neural networks with LSTM

In our approach we use an extension of Gated Graph Neural Networks – called here GGNN-LSTM – based on the Long Short-Term Memory (LSTM) cell [31]. This neural network architecture is based on the Gated Graph Neural Network architecture where the GRU memory cell is replaced with a LSTM cell. The overall architecture of the GGNN-LSTM is similar to the GGNN illustrated in Fig. 3.

Similarly to Eqs. (6)–(10), the propagation of the hidden representations among neighboring nodes for one time-step in a GGNN-LSTM is formulated as:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)}\left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \tag{11}$$

$$\mathbf{i}_v^{(t)} = \sigma\left(\mathbf{W}_i\mathbf{a}_v^{(t)} + \mathbf{U}_i\mathbf{h}_v^{(t-1)} + \mathbf{b}_i\right) \tag{12}$$

$$\mathbf{f}_v^{(t)} = \sigma\left(\mathbf{W}_f\mathbf{a}_v^{(t)} + \mathbf{U}_f\mathbf{h}_v^{(t-1)} + \mathbf{b}_f\right) \tag{13}$$

$$\mathbf{o}_v^{(t)} = \sigma\left(\mathbf{W}_o\mathbf{a}_v^{(t)} + \mathbf{U}_o\mathbf{h}_v^{(t-1)} + \mathbf{b}_o\right) \tag{14}$$

$$\mathbf{g}_v^{(t)} = \tanh\left(\mathbf{W}_g\mathbf{a}_v^{(t)} + \mathbf{U}_g\mathbf{h}_v^{(t-1)} + \mathbf{b}_g\right) \tag{15}$$

$$\mathbf{c}_v^{(t)} = \mathbf{c}_v^{(t-1)} \odot \mathbf{f}_v^{(t)} + \mathbf{g}_v^{(t)} \odot \mathbf{i}_v^{(t)} \tag{16}$$

$$\mathbf{h}_v^{(t)} = \tanh\left(\mathbf{c}_v^{(t)}\right) \odot \mathbf{o}_v^{(t)} \tag{17}$$

with $\{\mathbf{W}_i, \ldots\}$ and $\{\mathbf{U}_i, \ldots\}$ learnable weight matrices, and $\{\mathbf{b}_i, \ldots\}$ learnable bias vectors.

Eq. (11) is the propagation of the hidden representations among neighbors, as in Eq. (6). Eqs. (12)–(14) correspond respectively to the *input*, *forget* and *output* gates of the LSTM cell. Eq. (15) is a *candidate* hidden representation, with an initial value $\mathbf{c}_v^{(0)}$ set to zero. Eq. (16) is the internal memory of the LSTM cell.

Our rationale for using this alternate neural network architecture is motivated by better accuracy than the GGNN based on GRU presented in Section 3.2, as shown in the numerical evaluation in Section 5 for the TCP task.

3.4. Stacked gated graph neural networks

Since the neural network architectures introduced earlier are based on a special variant of recurrent neural networks, advances made for standard RNNs may also be applied here. For this purpose, we make use of stacked RNNs as proposed by Pascanu et al. [32], and illustrated in the following equation:

$$\mathbf{h}^{(t;l)} = r\left(A, \mathbf{h}^{(t-1;l)}, \mathbf{h}^{(t;l-1)}\right) \tag{18}$$

where $\mathbf{h}^{(t;l)}$ corresponds to the hidden representation at timestep t and layer l, and r a function describing the recurrent unit, i.e. Eqs. (6)–(10) for the GRU or Eqs. (12)–(14) for the LSTM unit.

3.4.1. Edge attention

A recent advance in neural networks has been the concept of *attention*, which provides the ability to a neural network to focus on a subset of its inputs. This mechanism has been used in a variety of applications such as computer vision or natural language processing (e.g. [33]). For the scope of GNNs, we introduce here so-called *edge attention*, namely we wish to give the ability to each node to focus only on a subset of its neighborhood. Formally, let $a_{(v,u)}^{(t)} \in [0, 1]$ be the attention between node v and u. Eq. (2) is then extended as:

$$\mathbf{h}_v^{(t)} = \sum_{u \in \text{NBR}(v)} a_{(v,u)}^{(t)} \cdot f^*\left(\mathbf{h}_u^{(t-1)}\right) \tag{19}$$

$$\mathbf{a}_{(v,u)}^{(t)} = f_A\left(\mathbf{h}_v^{(t-1)}, \mathbf{h}_u^{(t-1)}\right) \tag{20}$$

We decompose $f_A(\cdot)$ as two feed-forward neural networks and use an element-wise matrix multiplication to compute a modified adjacency matrix.

To make the computation of $a_{(v,u)}^{(t)}$ for all edges more efficient, we use the modified adjacency matrix $\widetilde{A}^{(t)}$ defined as:

$$\widetilde{A}^{(t)} = A \odot softmax\left(f_{A_1}\left(\mathbf{H}^{(t)}\right)^T \odot f_{A_2}\left(\mathbf{H}^{(t)}\right)\right) \tag{21}$$

with $f_{A_1}(\cdot)$ and $f_{A_2}(\cdot)$ feed-forward neural networks, and *softmax* the normalized exponential function such that:

$$softmax(\mathbf{X})_j = \frac{e^{X_j}}{\sum_i e^{X_i}} \tag{22}$$

4. Application to performance evaluations of networks

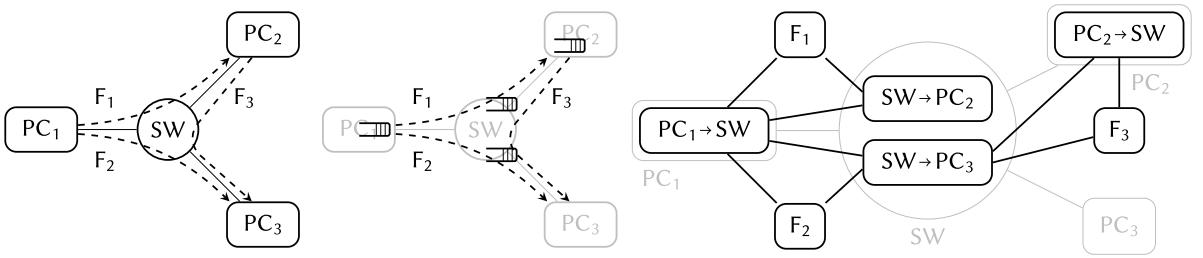
We describe in this section the application of the deep learning architectures presented earlier to the performance evaluation of network topologies and network protocols. In other words, our goal is to represent network topologies and the flows traversing them as graphs which can be passed as inputs to a GGNN. Compared to other works on the application of machine learning to performance evaluation, the main contribution is that this graph representation is a low-level input feature. This means that specific high-level features of the studied network protocol are not required and the trained machine learning algorithm is not restricted to a specific topology.

The main intuition behind the input feature modeling for the GGNN is to use the network of queues as input graph $\mathcal{G}$, with additional nodes representing the flows in this network. An illustration of this network of queues is given in Fig. 4b, which is the representation of the different queues in the example network depicted in Fig. 4a. Note that we represent on Fig. 4b only the forward path of the flows. Fig. 4b may also be extended to include the queues taken by the acknowledgment packets used by the flows if necessary, such as TCP ACK packets for example as explained later in Section 5.2.

We formally define here the transformation between an input computer network and the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ which will be analyzed by the GGNN. Fig. 4c is an application of this transformation applied to the network presented in Fig. 4a.

We focus in this article on the study of simple Ethernet networks, where computers are connected via simple store-and-forward switches and exchanging data via flows. We assume that each flow in the network is unidirectional, with a single source and a single destination. Each flow f is represented as a node v_f in the graph $\mathcal{G}$. As illustrated in Section 5.2, bidirectional flows such as TCP flows may still be analyzed by using two unidirectional flows, modeled in $\mathcal{G}$ by two nodes connected by an edge.

We assume here that the routing in the studied network is static, namely that the path taken by a flow is fixed. Each queue traversed by a flow in the network is represented as a node v_q in the graph $\mathcal{G}$. For simplifications purpose we only



(a) Physical network topology (b) Associated queuing network (c) Associated graph representation

Fig. 4. Graph encoding of an example network topology with three flows.

consider the queues present in the network interfaces and not other queues which may be present in operating systems. We model network interfaces using a simple drop-tail queue. In case of Ethernet switches, multiple nodes are used since multiple interfaces are present. Queues are connected by edges according to the physical topology of the studied network. Let Q_f be the set of queues traversed by flow f. In order to encode the path taken by a flow, edges connect the node representing the flow v_f with the nodes representing the traversed queues $\{v_q | q \in Q_f\}$.

Each node v in the graph $\mathcal{G}$ is characterized by its input features l_v, encoding parameters relevant to the studied network protocol. In the simple case illustrated here, we only encode the node type – *i.e.* if a node represents a flow or a queue – using one-hot encoding. Namely l_v is a vector with two values, with $[1, 0]^T$ is for queue nodes, and $[0, 1]^T$ for flow nodes.

Each node v in the graph $\mathcal{G}$ is also characterized by its output features o_v corresponding to the values which are predicted. In this article, we study in Section 5 the case where the output vector o_{v_f} of each flow f corresponds to a bandwidth or an end-to-end latency. Since we do not make prediction on queues here, their corresponding nodes have no output vector. This is implemented by using a masked loss function for training the neural network, which only take into account the output vectors of flows.

Note that for simplification purpose, we limited this section – and description of transformation between network and graph – to the case where all nodes in the network have the same behavior and that all links in the topology have the same capacity and latency. Additional features for distinguishing between different behaviors or node types (e.g. link capacity, different vendors, operating system) may easily be added in case differences between nodes are present and relevant to the prediction, such as different configurations, types or link capacities. In this case, l_v is extended with additional values representing those differences and encoded using standard practices for feature encoding in machine learning. Example of such scenario where each node has a unique attribute is given later in Section 5 where flow rates and packet sizes are additionally encoded in l_v.

We note that this network transformation process may easily be extended to more complex architectures and more refined models of queues, such as having network interfaces using multiple queues and a packet scheduler, or modeling more precisely network switches using additional queues depending on the internal architecture of switches. Since standard packet processing pipelines can easily be represented as graphs, additional nodes and edges representing more complex pipelines may be added in the graph.

Compared to more traditional approaches as in [2,3,8], we note that the graph and feature representations described here are independent of any studied network protocols or specific metrics (e.g. bandwidth, latency, etc.).

5. Numerical evaluation

In this section, we evaluate Graph Neural Networks in the context of our approach described in Sections 3 and 4 on two representative use-cases: prediction of the average steady-state bandwidth of TCP flows and prediction of end-to-end latencies of UDP flows.

5.1. Implementation

For both use-cases, the same implementation, parameters and learning algorithm for the neural network were used. The GGNN architectures presented in Sections 3.2 and 3.3 were implemented using Tensorflow [34] and trained using a Nvidia GeForce GTX 1080Ti GPU. The recurrent part of the GGNN and GGNN-LSTM were respectively implemented according to Eqs. (6)–(10) and Eqs. (11)–(17). The function $g(\cdot)$ in Eq. (4) was implemented using a feed-forward neural network with two dense layers. Additional dropout layers [35] were added according to standard practices for recurrent neural networks [36] in order to avoid over-fitting.

For each studied use-case, the model was trained multiple times using the parameters listed in Table 1. The learning rate used for training the models was optimized following standard practices based on Bayesian optimization. The learned weights producing the best result is then selected for the numerical results presented in the rest of this section. All GNN models evaluated here were trained for the same number of iterations.

Table 1
Parameters used for the training phase of the neural networks.

Parameter	Value
Size of hidden representation $\mathbf{h}_v$	96
Number of loops unrolling	12
Training algorithm	RMSProp [30]
Mini-batch size	64
Number of training iterations	20 000

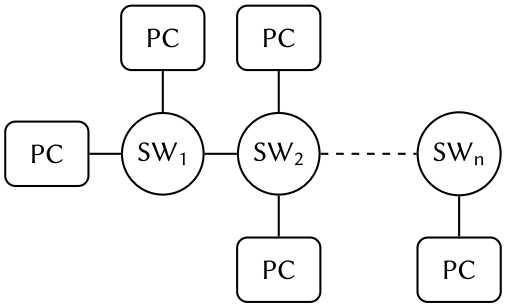


Fig. 5. Daisy chain topology used for the numerical evaluation.

The following Graph Neural Network architectures were evaluated:

- **GGNN-GRU:** A Gated Graph Neural Network using a GRU memory unit as presented in Section 3.2,
- **GGNN-LSTM:** A Gated Graph Neural Network using a LSTM memory unit as presented in Section 3.3,
- **GNN-RNN:** A simplified implementation of Graph Neural Network as described in Section 3.1 using a simple Recurrent Neural Network (RNN) architecture, where the hidden node representation is computed as:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)} \left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)} \right]^T + \mathbf{b}_a \tag{23}$$

$$\mathbf{h}_v^{(t)} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\mathbf{h}_v^{(t-1)} + \mathbf{b} \right) \tag{24}$$

  with $\mathbf{W}$ and $\mathbf{U}$ learnable weight matrices, and $\mathbf{b}$ a learnable bias vector.
- **GNN-RNN^2** and **GNN-RNN^3** as stacked versions of GNN-RNN following Section 3.4, with respectively 2 and 3 stacks, as indicating by the exponent in the architecture label.

In order to compare our approach with other machine learning approaches, we evaluate also the following additional models:

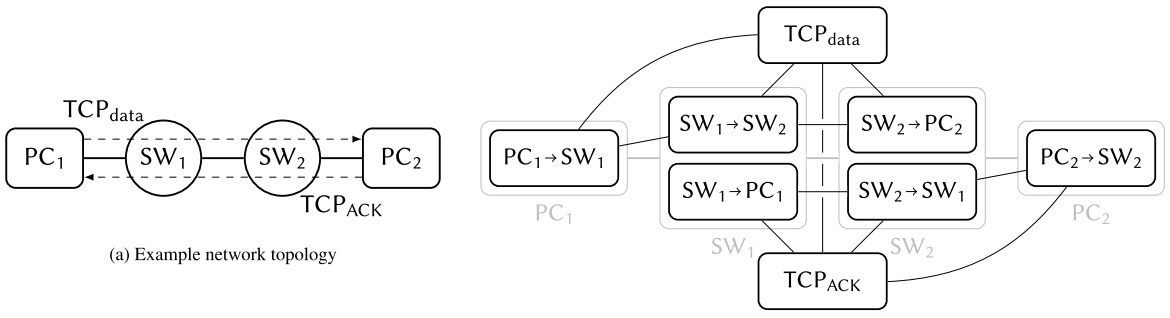
- **SVR:** Support Vector Regression for the TCP bandwidth predictions using measured round-trip time and loss probability as input features. This model is comparable to one proposed by Mirza et al. [2]. We note that the use of such model requires active measurement in the network. The implementation of SVR from `scikit-learn` [37] was used for the evaluation.
- **FFNN:** A feed-forward neural network with three dense layers for the TCP bandwidth predictions using the same features as the SVR model.

5.2. Use-case: prediction of average TCP flow bandwidths

In this first use-case, we evaluate the capabilities of our approach at predicting the steady-state bandwidth of TCP flows sharing different bottlenecks.

For the generation of the topologies, a random number of Ethernet switches is first selected using the discrete uniform distribution $\mathcal{U}(1, 6)$ and connected according to a daisy chain as illustrated in Fig. 5. A random number of nodes is then generated using the discrete uniform distribution $\mathcal{U}(2, 32)$ and connected to a randomly selected Ethernet switch. For each node, a TCP flow is generated with a randomly selected destination among the other nodes. The default parameters of *ns-2* for the TCP stack are used, meaning that TCP Reno is used as a congestion control algorithm. The results of the simulations are used as a basis for the learning process of the neural network.

Since the performance of TCP flows is highly dependent on the congestion experienced by TCP acknowledgments, we extend our approach from Section 4 to also include TCP ACKs in the input graph representation. Fig. 6 represents the graph encoding which has been used for this task on a simple topology. We follow the method explained in Section 4, where each queue in the network is represented by a node. Each TCP flows is encoded as two nodes in the graph: one node for representing the *data sub-flow* and one node for the *acknowledgment sub-flow*, both connected by an edge. Each node i in the graph has a feature vector encoding node type as a one-hot vector:



(a) Example network topology

(b) Associated graph encoding

Fig. 6. Graph encoding used for the TCP prediction task.

- $[1, 0, 0]^T$ for nodes representing queues,
- $[0, 1, 0]^T$ for TCP data nodes,
- $[0, 0, 1]^T$ for TCP acknowledgment nodes.

The output vector $\mathbf{y}_i$ of each node representing a TCP data sub-flow is a single value corresponding to the normalized steady-state bandwidth. The output vector of the other nodes is masked in the loss function.

As shown in previous studies about TCP [9,24], the average throughput of TCP flows depends on various parameters such as the TCP congestion control algorithm, the configuration of the TCP stack, round-trip times or drop probabilities, as well as the interaction between flows. The graph encoding used here offers a more low-level representation.

The neural network is trained against a mean-square loss function such that the following function is minimized:

$$\frac{1}{|flows|} \sum_{i \in flows} (\mathbf{y}_i - \mathbf{o}_i)^2 \tag{25}$$

5.3. Use-case: prediction of UDP flows end-to-end latencies

In this second use-case, we evaluate the capabilities of our approach at predicting the end-to-end latencies of UDP flows sharing different bottlenecks. We follow the same approach as for the previous use-case, namely multiple random topologies with UDP flows are generated and evaluated using simulations. The same daisy-chain topology presented in Fig. 5 is used here.

Each UDP flow i generates unidirectional traffic from a given pair of source S_i and destination D_i, with constant rate r_i, using packets of random size taken from a uniform distribution from 10 to m_i Bytes. The parameters S_i, D_i, r_i and m_i are randomly generated for each UDP flow. For this use-case we focus on the prediction of the average and 95% quantile of the end-to-end delay for each flow in a given topology.

Regarding the graph encoding, we follow the approach illustrated in Section 4 with Fig. 4 since flows are unidirectional. As for the TCP use-case, the feature vector of each node i encodes the node type as a one-hot value and the traffic parameters:

- $[1, 0, \widetilde{r_i}, \widetilde{m_i}]^T$ for nodes representing flows, with $\widetilde{r_i}$ and $\widetilde{m_i}$ the normalized versions of r_i and m_i
- $[0, 1, 0, 0]^T$ for nodes representing queues

The output vector $\mathbf{y}_i$ of each node representing a UDP flow corresponds to the normalized end-to-end latency. As for the TCP use-case, we use Eq. (25) as loss function, where the output vector of the non-flow nodes are masked.

5.4. Prediction accuracy

In order to evaluate the accuracy of the different models presented here, we evaluate the predictions using the relative absolute error and residual metrics:

$$\text{Relative absolute error:} \quad \frac{|\mathbf{y}_i - \mathbf{o}_i|}{\mathbf{y}_i} \tag{26}$$

$$\text{Residual:} \quad \mathbf{y}_i - \mathbf{o}_i \tag{27}$$

with $\mathbf{y}_i$ and $\mathbf{o}_i$ respectively measurements from the simulations, and predicted values.

Fig. 7 illustrates the distributions of relative error of the different neural network architectures for the datasets previously described.

We first focus our interpretation of Fig. 7 on the TCP use-case with machine learning models using high-level features (round-trip time and loss probability), namely SVR and FFNN. The median relative absolute error of 11.6% for the SVR model

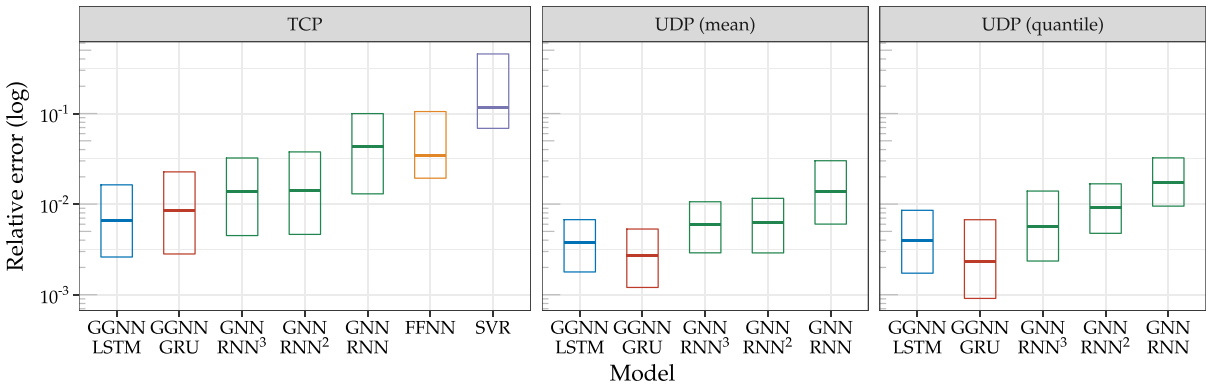


Fig. 7. Comparison of the evaluated machine learning methods according to the relative error. Bars represent respectively the 25, 50 and 75 percentile values.

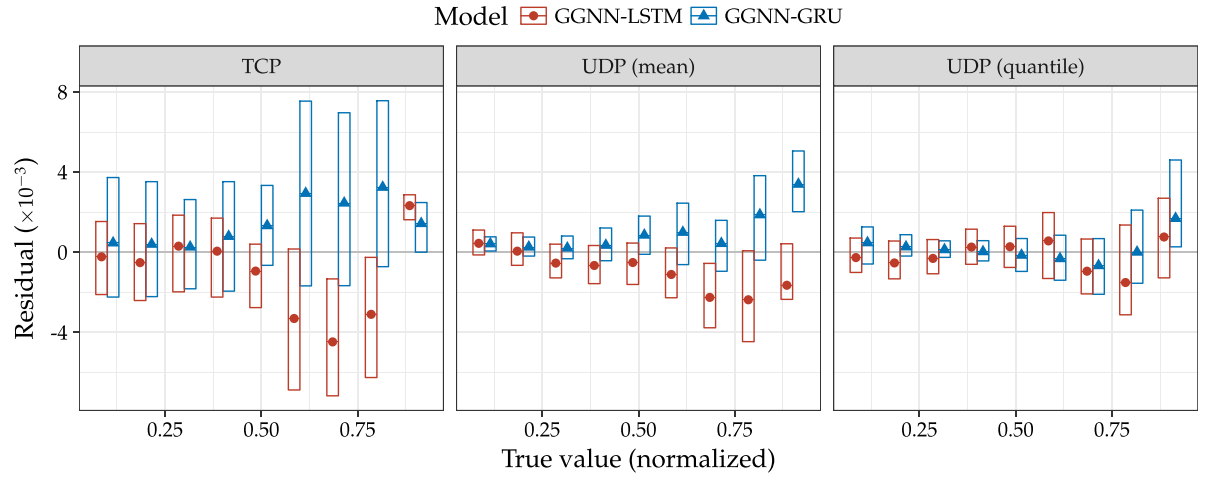


Fig. 8. Comparison of real value against the residual ($\mathbf{o}_i - \mathbf{y}_i$) for the GGNN-GRU and GGNN-LSTM architectures. Values are normalized according to the studied dataset. Bars represent respectively the 25, 50 and 75 percentile values.

is in line with the results from [2], which reported a 10% median relative error. The feed-forward neural network provides better results, with a median relative error of 3.5%. Those values will be used as a baseline for comparison purposes.

Regarding the GGNN models, all architectures evaluated here are able to predict the TCP bandwidths with a median relative error below 1%, outperforming the values from the SVR by one order of magnitude, and also outperforming the feed-forward neural network using high-level input features. This highlights our main motivation for using GGNNs.

Regarding UDP latencies, we are also able to reach a median relative error below 1%. The GGNN-LSTM architectures provide better results on the TCP use-case, while the GRU-based ones are more suited to the UDP use-case. We note that using stacked memory cells for the GGNN enables better accuracy for the TCP use-case, while not providing better results in the UDP use-case.

Compared to the more simpler GNN-RNN architecture, we notice that we reach better accuracy using a GRU or a LSTM memory cell, even in case of memory stacks. This motivates our choice of using more recent graph-based neural network architectures from [4] compared to the earlier results from [5,6].

In order to better understand the difference between both GRU- and LSTM-based architectures, we investigate the average residual as illustrated in Fig. 8.

For both use-cases we notice similar values for normalized values below 0.5, where both architecture result in similar residual. In the TCP use-case, both architectures provide similar results across the range of measured values, with a tendency for the GRU-based architecture to underestimate bandwidths, while the GGNN-based one overestimates bandwidths. A similar behavior is exhibited for the UDP use-case.

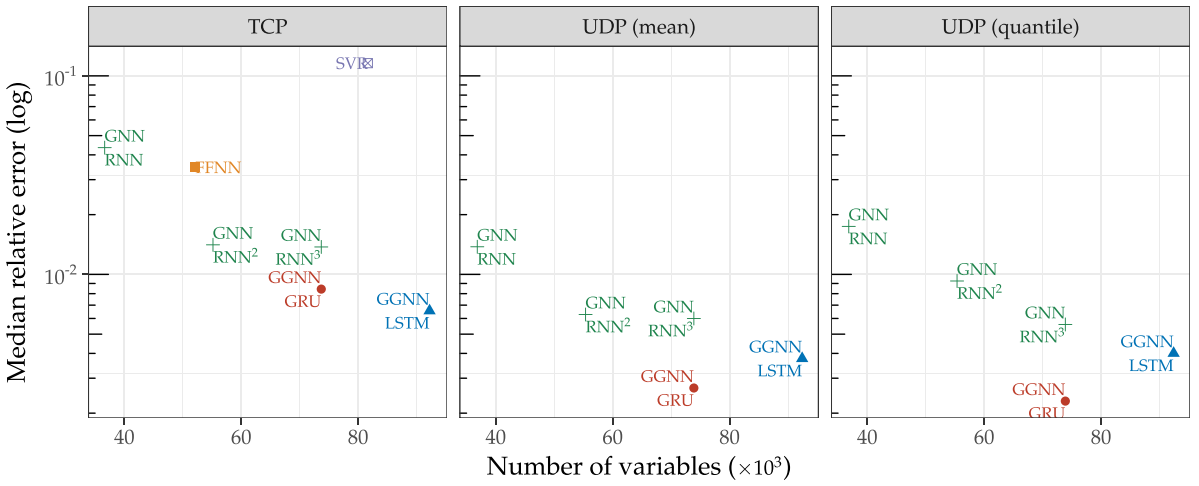


Fig. 9. Evaluation of the precision of the architectures against their number of variables.

5.5. Influence of model size

An important factor when comparing machine learning models and neural network architectures is their number of variables according to their precision, as illustrated in Fig. 9. For the SVR model, we quantify the number of variables as the size of the support vectors. For the neural network architectures, the number of variables presented in here are based on the same size for the hidden node representation (as highlighted in Table 1). Since model size directly correlates with training time and evaluation time, an ideal model would be in the bottom left part of Fig. 9, meaning it would provide the best accuracy with the lowest possible number of variables.

The difference between SVR and GGNN is also noticeable here, since both types of models use a similar number of variables, but with large differences in relative error. Fig. 9 illustrates also the motivation for using GGNNs with memory cells as the GGNN-GRU and GGNN-LSTM architectures outperform the other methods.

We also notice that although the GNN-RNN^3 and GGNN-GRU architectures have similar number of variables, the GGNN-GRU architecture outperforms the GNN-RNN^3 one, motiving our choice of more advanced graph-based neural network architecture.

5.6. Influence of graph size

We investigate in this section the relationship between size of the network topology and the precision of the model. Fig. 10 illustrates the influence of input graph size against the relative error. While some correlation between graph size and relative error are visible in Fig. 10, it is dependent on which dataset is investigated. We note that for the TCP use-case, increase in graph sizes result in an increase in the relative error. In the UDP use-case, the relationship between graph size and relative error is reversed.

5.7. Influence of number of unrolled loops

We noted in Section 3.2 that GGNNs are unrolled a fixed number of iterations T compared to GNNs. Fig. 11 illustrates the influence of T against the relative error, where the illustrated GGNNs were trained with different values for T. Since increasing T also has an impact on training and evaluation time, a trade-off between accuracy and speed has to be established when designing the graph neural network's architecture and choosing its hyper-parameters. Increasing T improves the prediction accuracy in both use-cases for the studied architectures, with smaller effect for T larger than 12, motivating our choice of $T = 12$ as presented in Table 1.

We notice that the UDP use-case highlights a larger difference between both architectures, where larger values of T for the GGNN-GRU provide only minimal impact on the prediction accuracy. This can be explained by the fact that end-to-end latencies depend more on local queuing effects than topology-wide ones. This is opposed to the performance of TCP flows, where the interaction between flows on a larger scale across the topology have larger impact than local effects.

5.8. Execution time

In order to understand the scalability of our model presented in Section 3, we evaluate in this section our implementation in terms of execution time. Since our datasets with TCP or UDP flows contain mainly small networks, we evaluated our

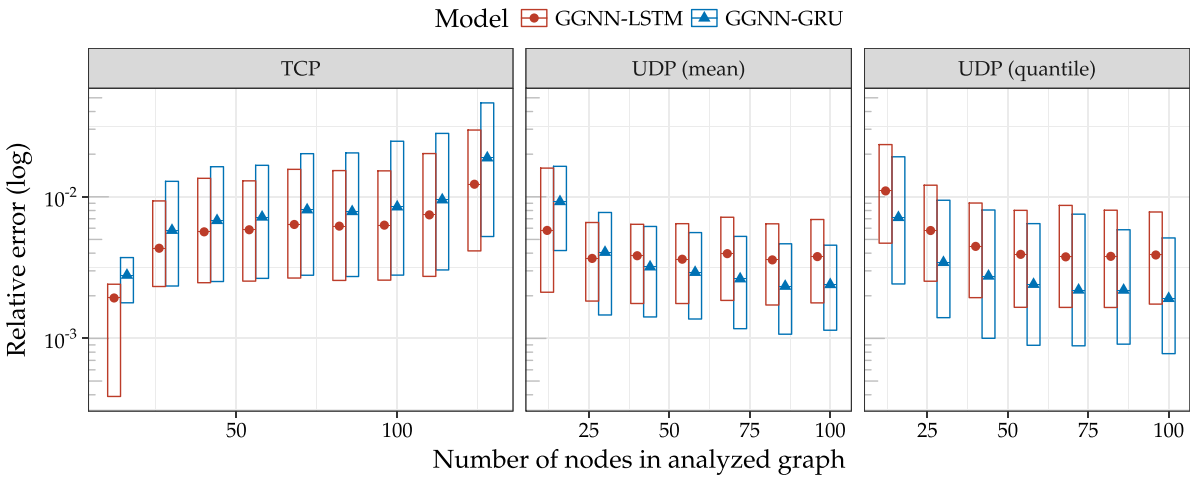


Fig. 10. Evaluation of architecture precision against the size of the studied graph for the GGNN-GRU and GGNN-LSTM models. Bars represent respectively the 25, 50 and 75 percentile values.

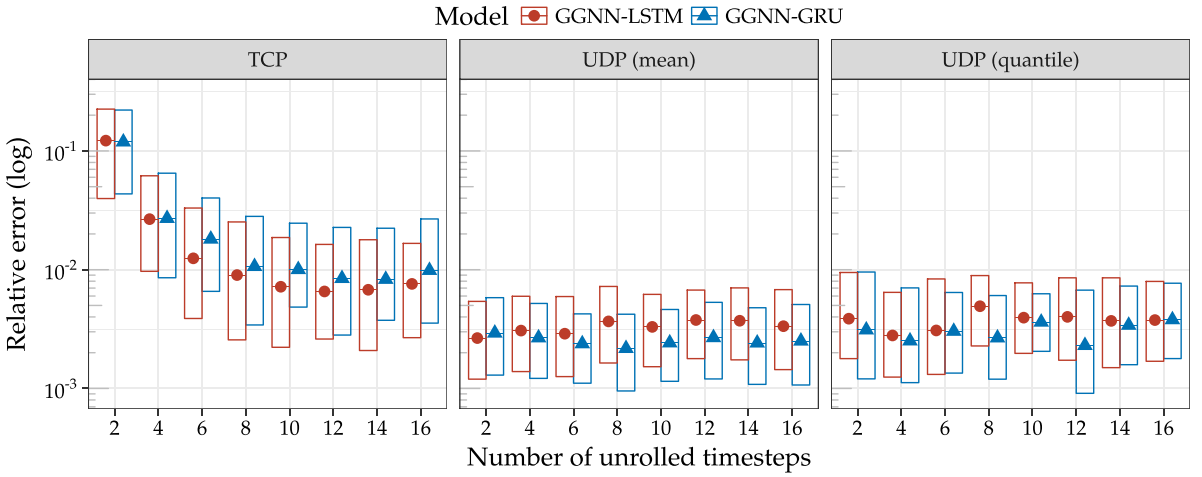


Fig. 11. Evaluation of architecture precision against the number of unrolled iterations of hidden representation propagation. Bars represent respectively the 25, 50 and 75 percentile values.

implementation against randomly generated graphs. We followed the Erdős–Rényi model [38] for generating random graph with a randomly selected number of nodes $|\mathcal{V}|$ following the uniform distribution $\mathcal{U}(2,\ 1000)$ and a random number of edges $|\mathcal{E}|$ following the uniform distribution $\mathcal{U}\ (N-1, D|\mathcal{V}|(|\mathcal{V}|-1)/2)$ with D the graph density parameter. The average graph density for both datasets used in Sections 5.2 and 5.3 was approximately 0.2.

Since the propagation of hidden node representations defined in Eqs. (2) and (6) may either be implemented in Tensorflow using dense matrix multiplication, or using sparse operations as noted by Allamanis et al. [20], we evaluated a dense and a sparse version of our model. According to the mathematical operations illustrated in Section 3 the runtime of one loop unrolling of the GNN correspond to the sum of the following terms:

- Per-node operations, namely Eqs. (7)–(10) for the GGNN-GRU, which scale linearly in execution time with the number of nodes, namely $\mathcal{O}\left(|\mathcal{V}|\right)$;
- The propagation of hidden node representations defined in Eqs. (2) and (6):
  - scaling linearly with the number of edges with sparse operations, namely $\mathcal{O}\left(|\mathcal{E}|\right)$, i.e. $\mathcal{O}\left(D|\mathcal{V}|^2\right)$ following that the $\max\left(|\mathcal{E}|\right) = D|\mathcal{V}|(|\mathcal{V}|-1)/2$,
  - scaling according to the multiplication of the hidden node representations with the graph adjacency matrix of size $|\mathcal{V}| \times |\mathcal{V}|$ with dense operations, namely $\mathcal{O}\left(\mathcal{H}|\mathcal{N}|^2\right)$ with $\mathcal{H}$ the size of hidden representation.

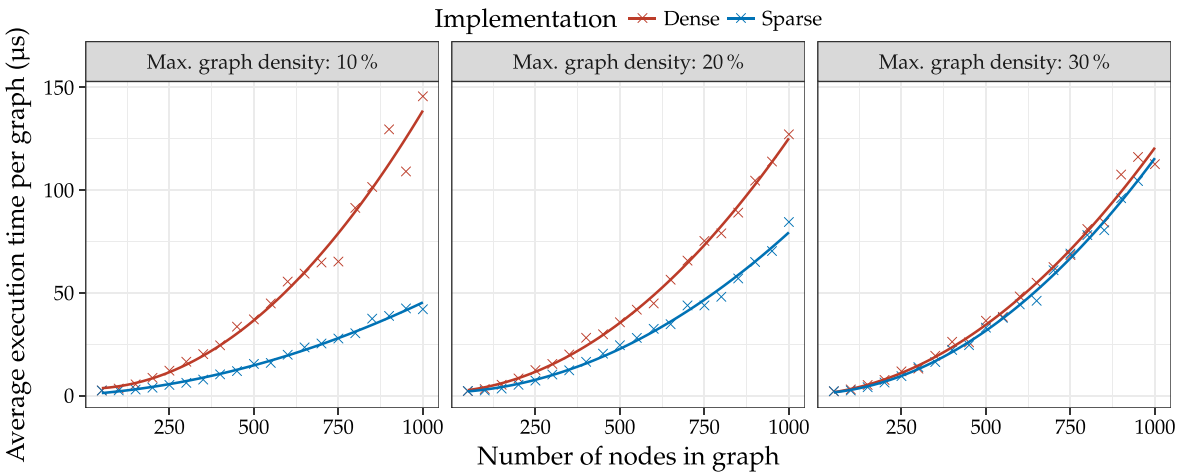


Fig. 12. Average execution time of prediction against number of nodes in graph. The fitted lines correspond to a polynomial model of order 2.

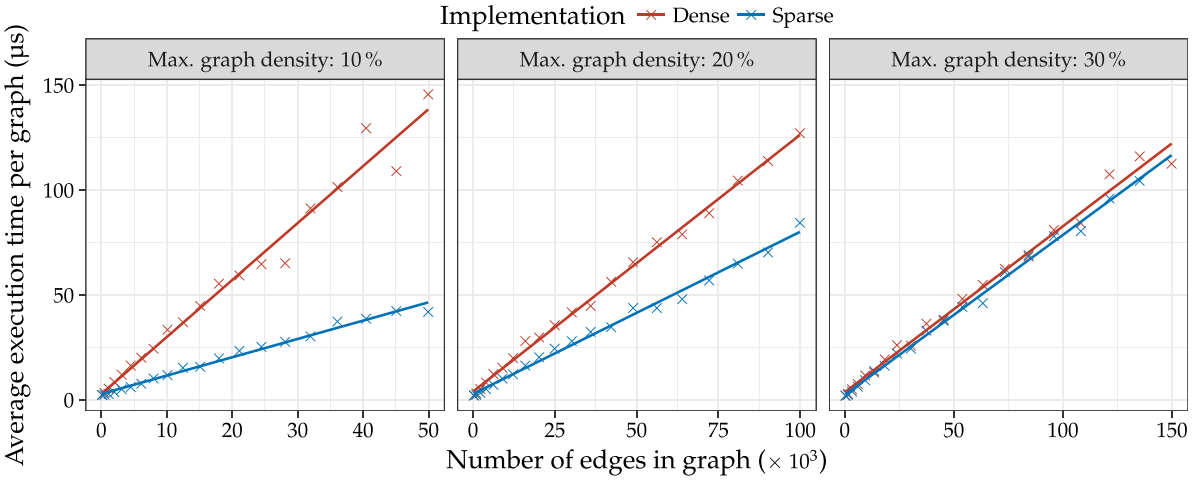


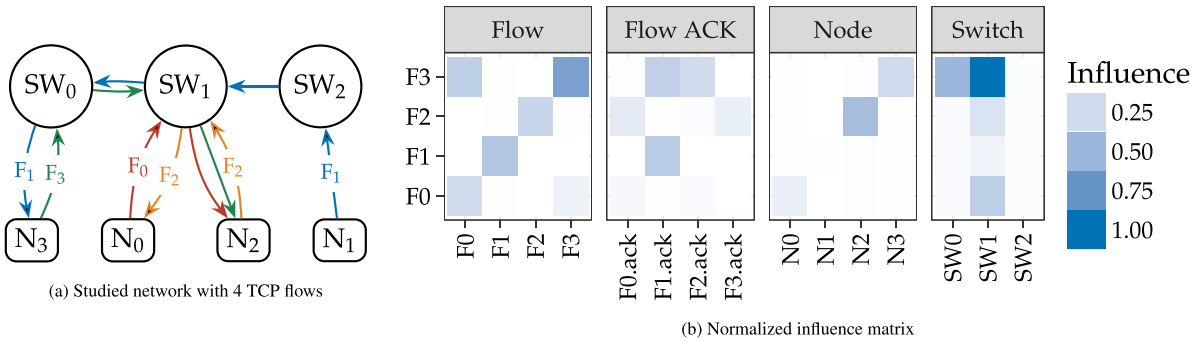
Fig. 13. Average execution time of prediction against number of edges in graph. The fitted lines correspond to a linear model.

Those operations are performed multiple times according to the number of loop unrolling, as discussed in Sections 3.2 and 5.7. In summary, our implementations should scale in execution time quadratically with the number of nodes in the graph, and linearly with the number of edges.

Numerical results are presented in Figs. 12 and 13 for our two different implementations and three values for the maximal graph density. The execution time measurements were measured on a server equipped with an Intel Xeon CPU E5-2620 v4 at 2.10 GHz and a Nvidia GeForce GTX 1080Ti. Batching of graphs was used to take advantage of parallelization. Those measurements validate the theoretical results presented earlier and illustrate the difference between the two implementations. As expected, the implementation using sparse operations performs faster for small graph densities while the implementation using dense operations performs the same regardless of graph density.

Those numerical results illustrate also applicability in practical use-cases since the execution time for prediction is below 200 us even for graphs with 1000 nodes and 150 000 edges. This small execution time enable fast operation in real networks, where the predictions may be used to dynamically configure computer networks.

Finally, while we illustrated in Figs. 12 and 13 the execution time for making a prediction on a given graph, our approach may still be limited by the size of the required dataset for larger networks in the training phase. Depending on the task which needs to be predicted, larger datasets may be required on larger graphs, since those larger cases may exhibit more complex behaviors. In the two use-case described in Sections 5.2 and 5.3, larger networks and larger number of flows will result in edge cases for the performance, namely low bandwidth for the TCP use-case and high latencies for the UDP one. Since those performance are inherently limited by link speed and maximum buffer sizes of the underlying topology in those edge cases, training and prediction on larger networks should still possible with moderate dataset sizes.



(a) Studied network with 4 TCP flows

(b) Normalized influence matrix

Fig. 14. Influence of nodes, other flows, and flow acknowledgments on the flows' performance.

6. Interpreting GNNs

An important subject when working with neural network is the interpretability of the learned model and its associated weights. A method often used for this purpose in computer vision is feature visualization using optimization [39], where inputs are optimized such that a specific output feature is maximized.

In our case, we are interested in visualizing the interaction between selected flows and the traversed queues. Once the parameters of the graph neural network learned, we replace the adjacency $\mathbf{A}$ with a modified matrix $\widetilde{\mathbf{A}}$ such that

$$\widetilde{\mathbf{A}} = \mathbf{A} \odot \sigma\left(\mathbf{B} + \mathbf{B}^T\right) \tag{28}$$

with $\mathbf{B} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ a learnable matrix. $\mathbf{B} + \mathbf{B}^T$ ensures that $\widetilde{\mathbf{A}}$ is a symmetrical matrix. For a given flow f, we then minimize the following loss function:

$$\left(\mathbf{y}_f - \mathbf{o}_f\right)^2 + \frac{1}{|\mathcal{V}|^2}\sum_{i,j}\widetilde{\mathbf{A}}_{\{i,j\}} \tag{29}$$

Fig. 14 illustrates the result of this optimization on a small topology with four TCP flows. We optimized $\mathbf{B}$ for each flow in the topology using gradient based optimization. The results are then aggregated according to the nodes in the network topology instead of the respective edges. The influence matrix corresponds to the aggregated value of $\mathbf{B}$.

As expected, bottlenecks and flows sharing the same bottleneck have an influence on each other, as shown for instance on the influence from SW_1 and F_0 on F_3. Nodes having no influence on the performance of flows can also be found, as illustrated here by SW_2 having no influence on F_1. Using this visualization method, root-cause analysis of poor performance and bottleneck identification may be performed.

7. Conclusion

We presented in this article *DeepComNet*, a novel approach for the performance evaluation of network topologies and flows based on graph-based deep learning. Our approach is based on the use of Gated Graph Neural Networks and a low-level graph-based representation of queues and flows in network topologies. Compared to other approaches using machine learning for performance evaluation of computer networks, the trained model is not specific to a given topology and high-level input features requiring more advanced knowledge on the studied protocol are not required.

We applied our approach to the performance evaluation of the bandwidth TCP flows and the performance evaluation of end-to-end latencies of UDP flows For both tasks, taking into account the topology in the performance evaluation is important: the throughput of TCP flows is dependent on the network architecture and network conditions (*i.e.* congestion and delays), and the end-to-end latencies depend on local queuing effects across the different queues from the topology. We showed via a numerical evaluation that our approach is able to reach good accuracies, with a median absolute relative error below 1%, even on large network topologies with multiple hops. We compared the chosen neural network architecture against two other approaches based on machine learning using high-level input features, and showed that our method outperforms those approaches by as much as one order of magnitude. Different types of GGNNs and configuration parameters were evaluated in order to understand which GGNN architecture produces the best results according to the studied network protocol. Finally we also gave some insights into the interpretation of the neural network in order to see which nodes in a given topology have an influence on protocol performances.

Since the network topology is directly taken as input of the neural network, applications such as network planning and architecture optimization may benefit from the method developed in this article. Future work may include further study of end-to-end protocols performance as well as optimization techniques based on this method.

Acknowledgments

We would like to thank Benedikt Jaeger for his feedback on an early draft of the paper as well as the anonymous reviewers for their valuable feedback. This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16KIS0538 (DecADe).

References

[1] G. Tian, Y. Liu, Towards Agile and smooth video adaptation in dynamic HTTP streaming, in: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, in: CoNEXT'12, ACM, ISBN: 978-1-4503-1775-7, 2012, pp. 109–120, http://dx.doi.org/10.1145/2413176.2413190.
[2] M. Mirza, J. Sommers, P. Barford, X. Zhu, A machine learning approach to TCP throughput prediction, IEEE/ACM Trans. Netw. (ISSN: 1063-6692) 18 (4) (2010) 1026–1039, http://dx.doi.org/10.1109/TNET.2009.2037812.
[3] M.B. Tariq, K. Bhandankar, V. Valancius, A. Zeitoun, N. Feamster, M. Ammar, Answering "What-If" deployment and configuration questions with WISE: techniques and deployment experience, IEEE/ACM Trans. Netw. (ISSN: 1558-2566) 21 (1) (2013) 1–13, http://dx.doi.org/10.1109/TNET.2012.2230448.
[4] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, in: Proceedings of the 4th International Conference on Learning Representations, in: ICLR'2016, 2016.
[5] M. Gori, G. Monfardini, F. Scarselli, a new model for learning in graph domains, in: Proceedings of the 2005 IEEE International Joint Conference on Neural Networks, in: IJCNN'05, vol. 2, IEEE, 2005, pp. 729–734, http://dx.doi.org/10.1109/IJCNN.2005.1555942.
[6] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Trans. Neural Netw. 20 (1) (2009) 61–80, http://dx.doi.org/10.1109/TNN.2008.2005605.
[7] Z.M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, K. Mizutani, State-of-the-art deep learning: evolving machine intelligence toward tomorrow's intelligent network traffic control systems, IEEE Commun. Surv. Tutor. (2017) http://dx.doi.org/10.1109/COMST.2017.2707140.
[8] H. Hours, E.W. Biersack, P. Loiseau, A causal approach to the study of TCP performance, ACM Trans. Intell. Syst. Technol. 7 (2) (2016) http://dx.doi.org/10.1145/2770878, 25:1–25:25.
[9] J. Padhye, V. Firoiu, D.F. Towsley, J.F. Kurose, Modeling TCP Reno performance: A simple model and its empirical validation, IEEE/ACM Trans. Netw. (ISSN: 1063-6692) 8 (2) (2000) 133–145, http://dx.doi.org/10.1109/90.842137.
[10] J. Bruna, W. Zaremba, A. Szlam, Y. LeCun, Spectral networks and locally connected networks on graphs, in: Proceedings of the 2nd International Conference on Learning Representations, in: ICLR'2014, 2014.
[11] M. Henaff, J. Bruna, Y. LeCun, Deep Convolutional Networks on Graph-Structured Data, arXiv:1506.05163.
[12] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, Computational capabilities of graph neural networks, IEEE Trans. Neural Netw. 20 (1) (2009) 81–102, http://dx.doi.org/10.1109/TNN.2008.2005141.
[13] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder–decoder for statistical machine translation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, 2014, pp. 1724–1734, http://dx.doi.org/10.3115/v1/D14-1179.
[14] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: Proceedings of the 5th International Conference on Learning Representations, in: ICLR 2017, 2017.
[15] M. Schlichtkrull, T.N. Kipf, P. Bloem, R.v.d. Berg, I. Titov, M. Welling, Modeling relational data with graph convolutional networks, in: Proceedings of the 15th European Semantic Web Conference, in: ESWC 2018, 2018.
[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, R. Pascanu, Relational inductive biases, deep learning, and graph networks.
[17] A. Grover, J. Leskovec, node2vec: Scalable feature learning for networks, in: Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 855–864, http://dx.doi.org/10.1145/2939672.2939754.
[18] D. Marcheggiani, I. Titov, Encoding sentences with graph convolutional networks for semantic role labeling, in: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2017, pp. 1506–1515, http://dx.doi.org/10.18653/v1/D17-1159.
[19] J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, G.E. Dahl, Neural message passing for quantum chemistry, in: D. Precup, Y.W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 1263–1272.
[20] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, in: Proceedings of the 6th International Conference on Learning Representations, in: ICLR 2018, 2018.
[21] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, D.L. Dill, Learning a SAT Solver from Single-Bit Supervision.
[22] M. Mathis, J. Semke, J. Mahdavi, T. Ott, The macroscopic behavior of the TCP congestion avoidance algorithm, ACM SIGCOMM Comput. Commun. Rev. 27 (3) (1997) 67–82, http://dx.doi.org/10.1145/263932.264023.
[23] N. Cardwell, S. Savage, T. Anderson, Modeling TCP latency, in: Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies, in: INFOCOM 2000, vol. 3, IEEE, 2000, pp. 1742–1751, http://dx.doi.org/10.1109/INFCOM.2000.832574.
[24] V. Firoiu, I. Yeom, X. Zhang, A framework for practical performance evaluation and traffic engineering in IP networks, in: Proceedings of the IEEE International Conference on Telecommunications, 2001.
[25] P. Velho, L.M. Schnorr, H. Casanova, A. Legrand, Flow-level network models: have we reached the limits?, Tech. Rep. 7821, INRIA, 2011.
[26] F. Geyer, S. Schneele, G. Carle, Practical performance evaluation of ethernet networks with flow-level network modeling, in: Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools, in: VALUETOOLS 2013, 2013, pp. 253–262, http://dx.doi.org/10.4108/icst.valuetools.2013.254367.
[27] F. Geyer, Performance evaluation of network topologies using graph-based deep learning, in: Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools, in: VALUETOOLS 2017, 2017, pp. 20–27, http://dx.doi.org/10.1145/3150928.3150941.
[28] L.B. Almeida, in: J. Diederich (Ed.), Artificial neural networks, IEEE Press, ISBN: 0-8186-2015-3, 1990, pp. 102–111.
[29] F.J. Pineda, Generalization of back-propagation to recurrent neural networks, Phys. Rev. Lett. 59 (1987) 2229–2232, http://dx.doi.org/10.1103/PhysRevLett.59.2229.
[30] T. Tieleman, G. Hinton, Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural Netw. Mach. Learn. 4 (2) (2012) 26–31.
[31] S. Hochreiter, J. rgen Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780, http://dx.doi.org/10.1162/neco.1997.9.8.1735.
[32] R. Pascanu, C. Gulcehre, K. Cho, Y. Bengio, How to construct deep recurrent neural networks, in: Proceedings of the 2nd International Conference on Learning Representations, in: ICLR 2014, 2014.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L.u. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, Vol. 30, Curran Associates, Inc., 2017, pp. 6000–6010.
[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, URL http://tensorflow.org/, Software available from tensorflow.org, 2015.
[35] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (1) (2014) 1929–1958.
[36] S. Semeniuta, A. Severyn, E. Barth, Recurrent Dropout without Memory Loss, arXiv:1603.05118.
[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[38] P. Erdős, A. Rényi, On random graphs. I, Publ. Math. 6 (1959) 290–297.
[39] M.D. Zeiler, R. Fergus, Visualizing and understanding convolutional networks, in: D. Fleet, T. Pajdla, B. Schiele, T. Tuytelaars (Eds.), Computer Vision – ECCV 2014, Springer International Publishing, ISBN: 978-3-319-10590-1, 2014, pp. 818–833, http://dx.doi.org/10.1007/978-3-319-10590-1_53.

Fabien Geyer is currently with Airbus Central Research & Technologies working on methods for network analytics, network performances and architectures. He received the master of engineering in telecommunications from Telecom Bretagne, Brest, France in 2011 and the Ph.D. degree in computer science from Technische Universität München (TUM), Munich, Germany in 2015. His research interests include formal methods for the performance evaluation and modeling of network architectures and protocols.