IFIP Networking 2019 1570511973

DeepMPLS: Fast Analysis of MPLS Configurations Using Deep Learning

Fabien Geyer
Technical University of Munich & Airbus CRT, Germany

Stefan Schmid
Faculty of Computer Science, University of Vienna, Austria

***Abstract*—With the increasing complexity of communication networks and the resulting threat of disruptions of mission critical services due to manual misconfiguration, automated verification is becoming a key element in today's network operation. In particular, it has recently been shown that a polynomial-time, automated verification of the policy-compliance of network configurations is possible for the important class of MPLS networks, even under failures. However, this approach, while providing polynomial runtimes, is still fairly slow in practice and only allows to *detect* but not *fix* configurations.**

This paper proposes a novel approach to speed up the analysis of network properties as well as to suggest configuration changes in case a network property is not satisfied. More specifically, our solution, *DeepMPLS*, allows to predict if a network property is satisfiable, and if not, aims to present a counter example. We also show that *DeepMPLS* may be used to propose new prefix-rewriting rules in the MPLS configuration in order to make it satisfiable. *DeepMPLS* can hence be used for fast predictions, before more rigorous analyses are performed.

***DeepMPLS* is based on a new extension of graph-based neural networks. Our prototype implementation, using Tensorflow, achieves low execution times and high accuracies in real-world network topologies.**

I. Introduction

As communication networks are increasingly used for critical services such as health monitoring, power grid management, or disaster response [1], their uninterrupted availability is more important than ever before. However, the increasing dependability requirements stand in stark contrast to today's manual approach to operate networks with often very complex configurations.

Automated approaches can greatly improve the trustworthiness of networks and hence reliability, by allowing to test a large number of network configuration for their policy compliance. Yet, many network verification tools still require a super-polynomial runtime to test basic connectivity properties [2, 3, 4]. Testing whether network configurations are policy compliant even under failures, introduces another combinatorial complexity.

It was recently shown that for the widely deployed MPLS networks, a polynomial-time "what-if analysis" is possible [5]: an automata-theoretic approach, leveraging a connection to prefix rewriting systems, can be used to test important properties such as connectivity (can two endpoints reach each other?), loop-freedom (may packets be forwarded in circles?) or waypoint enforcement (is traffic always going through the firewall?), even under failures. While this is promising, the runtime in practice is still relatively high (in the order of an hour even for relatively small yet complex networks): essentially, the approach in [5] requires the construction of a large pushdown automaton (PDA), based on the network configuration, the routing tables, as well as the query; the PDA is then solved using reachability analysis. While PDA is still polynomial in size, it can quickly grow to millions of nodes and transitions in realistic networks, for which reachability has to be solved *for each query*. Furthermore, the approach can only be used to *verify* properties, but not to *repair* configurations, e.g., to re-establish invariants.

This paper is motivated by the question whether it is possible to build upon these recent results while exploiting opportunities for speeding up verification as well as to support an automated fixing of configurations. This is challenging also because unlike other networks, MPLS supports arbitrary (and in principle unbounded) header sizes: additional labels are for example pushed to route around railed links. Our work is also motivated by a novel approach which seems to fit the specific problem particularly well: Graph Neural Networks [6, 7] have already been applied successfully in many contexts, including molecule analysis [8, 9] or jet physics [10], but despite being a natural choice for our problem, its potential is largely unexplored.

A. Our Contributions

This paper presents a novel approach to speed up verification and synthesis of the policy-compliance of network configurations. At the heart of our our tool, *DeepMPLS*, lies a new extension of graph-based neural networks: leveraging deep learning, *DeepMPLS* allows to predict counter examples (i.e., "proofs" or witness traces) to specific network properties (or queries), which can be verified fast. In fact, in this paper we show that *DeepMPLS*'s probabilistic approach may even be used for *synthesis*: it has the potential to predict which MPLS rules should be added, in order to *re-establish* certain properties. The tool may hence overcome the need to perform more rigorous and time-consuming analyses in many scenarios.

Our experiments, using our TensorFlow prototype implementation, show promising results: on real network topologies, *DeepMPLS* can achieve a high degree of accuracy with fast execution time.

ISBN 978-3-903176-08-9

As a contribution to the research community and in order to ease future research, our dataset and experimental data used in this paper is also available online.

B. Scope and State-of-the-Art

We are concerned with the correct, i.e., policy-compliant configuration of widely deployed MPLS networks. In particular, we are interested in predicting and fixing *properties* of MPLS networks which are described as a regular query language, as it is also used in the state-of-the-art *P-Rex* tool [4] motivating our paper. To this end, a formal model for MPLS networks is required. Before we sketch the model (see [4] for more details), we briefly review some basic concepts of MPLS networks.

In a nutshell, MPLS networks operate between Layer 2 and Layer 3 and rely on tunnels across a transport medium. Forwarding decisions are based on the *top-of-stack label*: an MPLS node (i.e., a.k.a. label switch router a.k.a. transit router) uses the top label of the label stack included in the packet header, to determine the next hop on a Label Switched Path (LSP). On this occasion, the old label can be replaced with a new label before the packet is routed forward. An MPLS node serving as *label edge router* acts as the entry and exit point for the network: a label edge router pushes an MPLS label onto an incoming packet resp. pops it from an outgoing packet.

More specifically, upon receiving a packet and depending on the content of the top of the stack label, an MPLS node performs a *swap*, *push* or *pop* operation on the packet label stack: In a *swap* operation the label is swapped with a new label, and the packet is forwarded along the path associated with the new label; in a *push* operation a new label is pushed on top of the existing label, encapsulating the packet in another layer of MPLS and introducing hierarchical routing; and in a *pop* operation the label is removed from the packet. If the popped label was the last on the stack, the packet leaves the MPLS tunnel.

In order to deal with failures, MPLS includes a local protection mechanism allowing to protect a label switched path by a backup path. This mechanism is based on the recursive pushing of labels, i.e., tunnels, around a failed link.

Our work is motivated by the goal to predict and fix properties according to a natural regular query language [4]. A (reachability) query is of the form $< a > \ b \ < c > \ k$ where the regular expression a describes the (potentially infinite) set of allowed initial label-stack headers, the regular expression b describes the set of allowed routing traces through the network, and the regular expression c describes the set of label-stack headers at the end of the trace. Finally, k is a number specifying the maximum allowed number of failed links.

This query needs to be answered for a given MPLS network whose configuration can be described as a tuple $N = (V, I, L, E, \tau)$: V is the set of routers, I the set of all interfaces in the network connected by links E, L the set of label stack symbols and τ the routing table (including conditional failover rules).

P-Rex allows to answer the following question in polynomial time: is there a set of failed links F with $|F| \leq k$ for a given network configuration $N = (V, I, L, E, \tau)$ such that there results a route (i.e., *trace*) satisfying the regular expression?

Towards this goal, P-Rex automatically collects the current routing tables, and given them as well as the network and the query, constructs a pushdown automaton (PDA) on which reachability analysis is performed using the Moped tool. More specifically, the initial header and final header regular expressions of the query are first converted to a Nondeterministic Finite Automaton (NFA) and then to a Pushdown Automaton (PDA). The path query is converted to an NFA, which is used to augment the PDA constructed based on the network model. The three PDAs are combined into a single PDA which simulates the automata running in lockstep and can then be queried.

P-Rex then not only provides a yes or no answer, but also a witness, if it exists. Furthermore, P-Rex additional optimizations such as "top of stack reduction", reducing the number of transitions in the PDA.

Thus, the challenge considered in this paper is to not only reduce the runtime further (which keeping the support for arbitrary header sizes and multiple link failures), using a novel methodology, but also to again "*synthesize*" a witness for each query. In particular, we would like to improve the runtime for "hard queries": to this end, we propose a methodology which considers the size of the PDA as a measure of complexity

C. Organization

The remainder of this paper is organized as follows. In Section II, we present our approach and solution in detail. Section III describes the dataset used for training and evaluating our approach, and Section IV reports on our experimental results. After reviewing related work in Section V, we conclude our work in Section VI and discuss future work.

II. DeepMPLS based on Graph Neural Network

This section presents our approach, *DeepMPLS*, supporting the fast testing and synthesis of MPLS network configurations. The main idea behind *DeepMPLS* is to map network topologies to graphs, which can then be processed using neural networks. Our graph representation has nodes representing routers, physical interfaces of routers and additional nodes used for describing MPLS configurations and queries. Edges between the nodes represent physical links in the topology, as well as the interactions between a MPLS configuration and elements of the network topology. Those graphs are then used as input for a neural network architecture able to process general graphs.

Compared to other representations used in machine learning which require to summarize properties of a topology in a vector of fixed size, our approach is not limited by the size of the topology or its configuration. This means that an accurate description of the complete topology and its configuration can be passed to the neural network.

A. Leveraging Graph Neural Networks

The neural network architecture used by *DeepMPLS* is based on an extension of Graph Neural Networks [6, 7]. In the following, let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. Let $\mathbf{i}_v$ and $\mathbf{o}_v$ represent respectively the input features and target values of node v.

Graph Neural Networks rely on a *message passing* concept:

$$\mathbf{h}_v^{(t)} = f\left(\left\{\mathbf{h}_u^{(t-1)} \;\middle|\; u \in \text{NBR}(v)\right\}\right) \quad (1)$$

$$\mathbf{o}_v = g\left(\mathbf{h}_v^{(t\to\infty)}\right) \quad (2)$$

$$\mathbf{h}_v^{(t=0)} = init\left(\mathbf{i}_v\right) \quad (3)$$

with $\mathbf{h}_v^{(t)}$ corresponding to the hidden representation of node v at time t, $f(\cdot)$ a function which aggregates the hidden representations, $\text{NBR}(v)$ the set of neighboring nodes of v, $g(\cdot)$ a function transforming the final hidden representation to the target values, and $init(\cdot)$ an initialization function for the hidden representations.

The concrete formulations of the $aggr$ and out functions are feed-forward neural networks (FFNN), with the addition that $aggr$ is the sum of per-edge terms [7], such that:

$$\mathbf{h}_v^{(t)} = aggr\left(\left\{\mathbf{h}_{\text{NBR}(v)}^{(t-1)}\right\}\right) = \sum_{u\in\text{NBR}(v)} f\left(\mathbf{h}_u^{(t-1)}\right) \quad (4)$$

with f a FFNN. $init$ is modeled as a one-layer FFNN which produces a vector respecting the dimensions of the hidden representations.

Gated Graph Neural Networks (GGNN) [11] were recently proposed as an extension of GNNs to improve their training. This extension implements f using a memory unit called Gated Recurrent Unit (GRU) [12] and unrolls Equation (1) for a fixed number of iterations. This simple transformation allows for commonly found architectures and training algorithms for standard FFNNs as applied in computer vision or natural language processing.

Formally, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{x}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{A} + \mathbf{b}_a \quad (5)$$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_z x^{(t)} + \mathbf{U}_z \mathbf{H}^{(t-1)} + \mathbf{b}_z\right) \quad (6)$$

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}_r x^{(t)} + \mathbf{U}_r \mathbf{H}^{(t-1)} + \mathbf{b}_r\right) \quad (7)$$

$$\widetilde{\mathbf{H}}^{(t)} = \tanh\left(\mathbf{W}x^{(t)} + \mathbf{U}\left(\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)}\right) + \mathbf{b}\right) \quad (8)$$

$$\mathbf{H}^{(t)} = \left(1 - \mathbf{z}^{(t)}\right) \odot \mathbf{H}^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{H}}^{(t)} \quad (9)$$

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 trainable weight matrices, and $\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}$ are trainable bias vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix, determining the edges in the graph $\mathcal{G}$.

Equation (5) corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for GNNs in Equations (1) and (4). Equations (6) to (9) correspond to the mathematical formulation of a GRU cell [12], with Equation (6) representing the GRU reset gate vector, Equation (7) the GRU update gate vector, and Equation (9) the GRU output.

B. Application to MPLS Network Analysis

In order to tailor the above concepts to MPLS network verification and synthesis, we need a transformation of network topology and MPLS configuration to a graph. The transformation process we propose in this paper is illustrated in Figures 1 to 4, where the MPLS network depicted in Figure 1 is transformed into a graph.

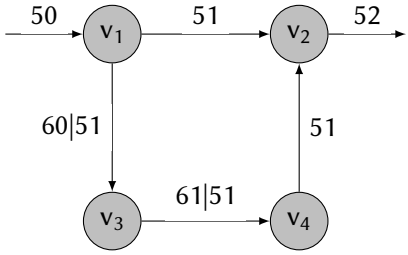

Figure 1: Example MPLS network. In case the link between v_1 and v_2 fails, a backup tunnel (v_1, v_3, v_4) has is used around the failed link.

Each router $v \in V$ in the network is represented as a node in the graph $\mathcal{G}$. Each network interface $i \in I_v^{in} \cup I_v^{out}$ is also represented as a node, connected via an edge to its router. Links in the topology E are represented as edges connecting the two corresponding network interfaces.

As presented in Figures 2 and 3, the MPLS configuration of each router is also encoded as nodes and edges in the graph. Each MPLS label $l \in L$ is represented as a node. The routing table of each router $\tau_v : I_v \times L \to \left(2^{I_v \times Op^*}\right)^*$ is represented as a set of rules. Each rule is represented as a node in the graph, connected the nodes representing its input interface $i \in I$ as well as its input label $l \in L$. The actions $o \in Op$ associated to a rule are also represented as nodes, connected via edges in case multiple actions are to be performed for a given rule as illustrated in Figure 3. MPLS actions with label parameters such as *swap* or *push* are connected to their respective label node. The last action associated to a rule is connected to its output interface.

Queries are also encoded as nodes in the graph as illustrated in Figure 4. In this paper, we will assume the same notation as [4] for specifying queries, namely the regular expressions which are defined over an alphabet Σ and use the abstract syntax

$$a ::= s \mid . \mid [\hat{}\, s_1, \ldots, s_n] \mid a_1 + a_2 \mid a_1 a_2 \mid a^*$$

where

s	is a symbol from Σ,
$.$	is a wildcard for any symbol from Σ,
$[\hat{}\, s_1, ..., s_n]$	stands for any symbol $s \in \Sigma \setminus \{s_1, \ldots, s_n\}$,
$a_1 + a_2$	is the choice between a_1 and a_2,
$a_1 a_2$	is the concatenation of a_1 and a_2, and

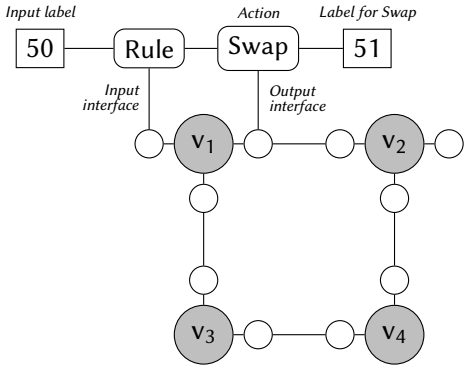

Figure 2: Encoding of network topology and a MPLS rule for v_1 as graph.

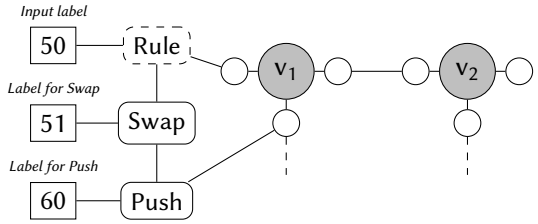

Figure 3: Encoding of network topology and a second MPLS rule for v_1 as graph.

a^* is the concatenation of 0 or more occurrences of a.

The set of all regular expressions over Σ is denoted by $Reg(\Sigma)$ and we assume a standard definition of the language $Lang(a) \subseteq \Sigma^*$ that is described by a regular expression a.

We follow an approach inspired by the McNaughton-Yamada-Thompson algorithm [13] which transforms a regular expression into an equivalent nondeterministic finite automaton. The different symbols of the regular expression of a query are represented as nodes, with edges representing their relationships. In case a symbol corresponds to a MPLS label or a router in the network, we reuse the node which was already defined in the graph. Wildcard symbols are represented as special nodes in the graph as illustrated in Figure 4.

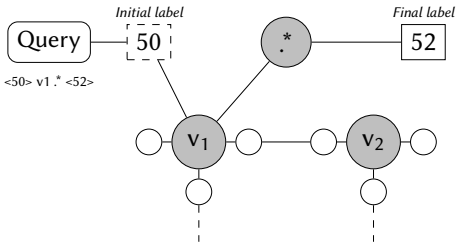

Figure 4: Encoding of network and query as graph.

Relationship between symbols such as combinations ($a_1 + a_2$) are represented using edges in the query representation, as illustrated in Figure 5.

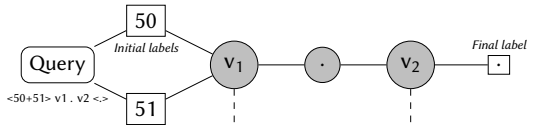

Figure 5: Encoding of more complex query as graph.

Each node in the graph may have input features describing characteristics of the node. In our case, nodes are mainly represented by their type, encoded as categorical value. We define the 12 following types for the nodes:

- Elements of the network topology: *Router*, *Interface*;
- Elements of the MPLS configuration: *Rule*, *Label*, *Push Action*, *Swap Action*, *Pop Action*;
- Elements of the query and the regular expression: *Query*, *Label Wildcard*, *Label Dot*, *Router Dot*, *Router Wildcard*.

Additionally to the type, the node representing a query has an additional input feature corresponding to the k parameter. Edges in the graph have no input features and represent only the relationship between nodes.

For training of the graph neural network, we use different output features depending on the prediction which is required. We define the three following prediction tasks:

Satisfiability Heuristic for verifying if a query is satisfiable.

Routing trace Heuristic for generating a trace of routers which match a satisfiable query.

Partial synthesis Synthesis of an MPLS configuration in order to satisfy a query.

Example queries for those three tasks are detailed later in Section III.

For the *Satisfiability* task, a classification task is defined where the query node is classified in two categories (*true* or *false*) in case the query is satisfiable or not. The training is done using a softmax cross entropy loss on the query node.

Similarly, for the *Routing trace* task, router nodes are classified in two classes, namely if the router is part of the trace or not. Since we are interested in the per-topology prediction, namely correctly classifying the router nodes in the graph, the training is done using a graph-wide sigmoid cross entropy loss.

Finally for the *Partial synthesis* task, missing rule and action nodes are added in the graph with the goal of connecting them via edges to the appropriate router, label and interfaces in order to satisfy a query. This case is an edge prediction task. Predicted edges in the graph are then used for reconstructing the MPLS prefix-rewriting rules. A similar loss function than for the *Routing trace* task is used here.

C. Complexity Analysis

In order to understand the scalability of the model presented in Section II-A, we evaluate the complexity of the algorithm. According to the mathematical operations illustrated earlier, the runtime of one loop unrolling of the GNN corresponds to the sum of the following terms:

- Per-node operations, namely Equations (6) to (9), which scales linearly in execution time with the number of nodes, namely $\mathcal{O}(|\mathcal{V}|)$;

- The propagation of hidden node representations defined in Equations (1) and (5), which scales linearly with the number of edges, namely $\mathcal{O}(|\mathcal{E}|)$ if sparse operations are used (i.e. using the graph's adjacency list), or quadratically to the number of nodes if the graph's adjacency matrix is used instead $\mathcal{O}(|\mathcal{V}|^2)$.

In order to achieve good accuracy, the recursion from Equation (1) is unrolled for a fixed number of iterations according to $\mathcal{D}$, the diameter of the analyzed graph. In total, the runtime complexity is summarized as: $\mathcal{O}\left(\mathcal{D}(|\mathcal{E}| + |\mathcal{V}|)\right)$.

III. Methodology and Dataset Generation

In order to train our neural network architecture, we used the Topology Zoo [14] – a collection of over 250 networks used in real-file – as a basis for generating network topologies. Each network topology was either taken as is or randomly modified, either by removing a router, or by adding a router and connecting it to a set of randomly chosen routers.

Based on those network topologies, MPLS configurations were generated for each network topology. With a given probability, MPLS tunnels were constructed between randomly selected pairs of routers in the network using dedicated MPLS labels. Additionally to the tunnel corresponding to the shortest path in the topology, a random number of additional backup tunnels were also generated following different paths in the network when possible. MPLS label stacking was used in for those backup tunnels, following common practice for automatic fail-over.

For generating queries, we randomly generate regular expressions as follows, with l_i representing the input label, l_o to output label, r_i the input router, r_o the output router, and k the maximum allowed number of failed links:

- $< l_i > r_i < l_o > k$ is satisfied if a packet with label l_i, crosses router r_i, and exists with label l_o;
- $< l_i > r_i \ .^* \ r_o < l_o > k$ is satisfied if a packet with label l_i entering at router r_i, traverses an unknown number of other routers, and exists from router r_o with label l_o;
- $< l_i > . \ .^* \ r_o < l_o > k$ is satisfied if a packet with label l_i enters the network, traverses at least one router, and exists from router r_o with label l_o;
- $< .^* > r_i \ .^* \ r_o < l_o > k$ is satisfied if there is a path from router r_i to r_o where the output label is l_o;
- $< l_i > r_i \ .^* \ r_o < .^* > k$ is satisfied if a packet with label l_i entering at router r_i, traverses the network, reaches router r_o with label l_o.

Those queries, inspired by the ones presented in [4], mainly focusing on reachability.

Routers and labels are either select randomly in the set of available routers V and labels L – resulting in most cases in non-satisfiable queries – or they are selected such that the query is satisfiable. In order to generate those satisfiable queries, we construct a so-called *MPLS traversal graph* for each network topology. In such a directed tree, each node is a tuple of the form $(\mathit{input\ label}, \mathit{router}, \mathit{output\ label}(s))$ corresponding to the result of the MPLS routing table of each router in the topology. Based on those MPLS traversal graphs, satisfiable queries can be generated by randomly selecting a node in the graph and traversing it randomly. Finally the k parameter of the query is generated randomly following a discrete uniform distribution.

The satisfiability of each generated query is tested using P-Rex [4], which relies on the construction of a push-down automaton based on the query as well as the network. Concretely, P-Rex generates one big pushdown automaton (PDA) based on the regular expressions of the initial header and final header defined by the query (which are first converted to nondeterministic finite automata) as well as the nondeterministic finite automata describing the path query.

Typically, the larger the PDA, the higher is the runtime of reachability analysis, and accordingly, in our methodology in the following, we will interpret the size of the PDA as a measure of complexity. Accordingly, for each evaluation of P-Rex, the size of the generated push-down automaton is recorded and will serve as a numerical measure of the complexity of the query in Section IV.

P-Rex can directly be used for defining the required outputs of the *Satisfiability* and *Trace* tasks. For the *Partial synthesis* task, we first randomly generate a satisfiable query and randomly remove a maximum of two MPLS prefix-rewriting rules such that the query is not satisfiable anymore. The rules which trigger the loss of satisfiability are the rules that *DeepMPLS* has to predict.

In total, more than 90.000 topologies and queries were generated. Table I summarizes different statistics about the generated dataset. The dataset is available online[1] to reproduce the results.

Parameter	Min	Max	Mean	Median
# of routers	3	30	10.6	10
# MPLS labels	8	689	225.3	174
# MPLS rules	8	795	319.5	248
Size of push-down automaton	17	37006	5441.2	2692
# of nodes in analyzed graph	36	2333	914.4	713
# of edges in analyzed graph	48	4000	1615.4	1261

Table I: Statistics about the generated dataset.

IV. Evaluation

We evaluate in this section *DeepMPLS* on the three prediction tasks described in Section II-B against our dataset of topologies and queries. Following current best practices for machine learning, the dataset was randomly split in two parts: training ($80\,\%$ of the topologies) and test ($20\,\%$ of the topologies). The neural network was trained on the training dataset, while the evaluation and metric figures reported later in this section were computed using the test dataset. Due to the lack of availability of other topologies and their MPLS configuration, no validation dataset was used.

Via a numerical evaluation, we illustrate the accuracy and execution time of *DeepMPLS* and highlight its usability

[1] https://github.com/fabgeyer/dataset-networking2019

for practical use-cases. The performances are also compared against a simpler heuristic based on a random walk in the MPLS network, and random prediction of MPLS rules to add to the configuration.

A. Technical Implementation

We implemented *DeepMPLS* using TensorFlow. For the purpose of computational efficiency, sparse operations are used for passing of hidden representation between neighboring nodes. Table II illustrates the size of the different layers of the neural network used for the numerical evaluation.

The recursion from Equation (1) was unrolled for a fixed number of iteration according to the diameter of the analyzed graphs. A detailed evaluation of the importance of this number of iterations will be performed in Section IV-F.

Layer	NN Type	Size
init	FFNN	$(14, 80)_w + (80)_b$
Memory unit	GRU cell	$(160, 160)_w + (160, 80)_w + (240)_b$
Edge attention	FFNN	$(160, 1)_w + (1)_b$
out hidden layers	FFNN	$2 \times \{(80, 80)_w + (80)_b\}$
out final	FFNN	$(80, 2)_w + (2)_b$
	Total:	53 124 parameters

Table II: Size of the different layers used in the GGNN. Indexes represent respectively the weights (w) and biases (b) matrices.

B. Random Walk Baseline

In order to have a baseline for evaluating the accuracy of the predictions of the neural network for the *Satisfiability* and *Trace* tasks, we introduce here a simple heuristic based on random walks in the MPLS network.

This heuristic selects a starting router and label in the topology according to the first elements of the query, and traverses the network according to the MPLS prefix-rewriting rules until the destination specified by the query is reached. In case multiple rules apply to a given input label, a random rule is selected and its prefix-rewriting actions are applied. If the random walk was not successful, another random walk is performed until a maximum number of walks of 10 is reached.

Since our dataset also contains queries which do not explicitly specify the starting label or starting node, as explained in Section III, a random starting point in the network is selected which matches the explicit parts of the query in those cases.

On the generated dataset, this heuristic was able to predict the satisfiability of a query with an accuracy of 79.2 %. As illustrated later in Figure 8, this accuracy decreases with the complexity of the query.

C. Neural Network Training

We first evaluate the training of *DeepMPLS* for the *Satisfiability* task, namely prediction of the satisfiability of a query given an topology and MPLS configuration. Figure 6 illustrates the accuracy of *DeepMPLS* during training according to the number of training iterations, on both the training and the test dataset. Each training iteration corresponds to 16 analyzed topologies and queries from the training dataset, i.e. their representations as graphs. After 2500 training iterations, *DeepMPLS* reaches the accuracy of the baseline on the test dataset, before converging after around 25 000 training iterations.

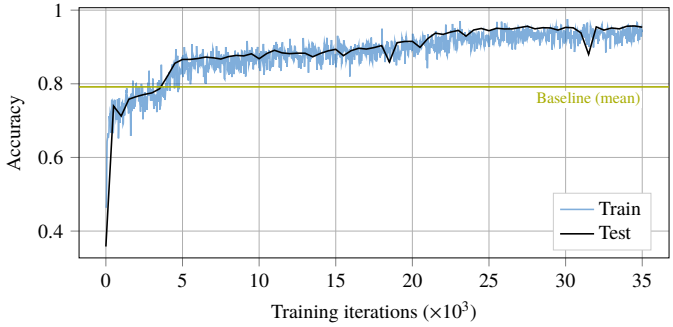

Figure 6: Training of *DeepMPLS* for prediction of query satisfiability and comparison against baseline.

Based on this first training, we retrain the same neural network and same set of weights for the *Routing trace* task, namely prediction of the routing trace of a query in case a query is satisfiable. This technique, also known as *knowledge transfer*, is often used in deep learning in order to accelerate training.

Figure 7 illustrates the accuracy of *DeepMPLS* on this second task according to the number of training iterations. Since the neural network was already pre-trained on the first task, this second training requires fewer iterations in order to reach good prediction accuracy. Less than 1000 iterations are required before convergence of the accuracy.

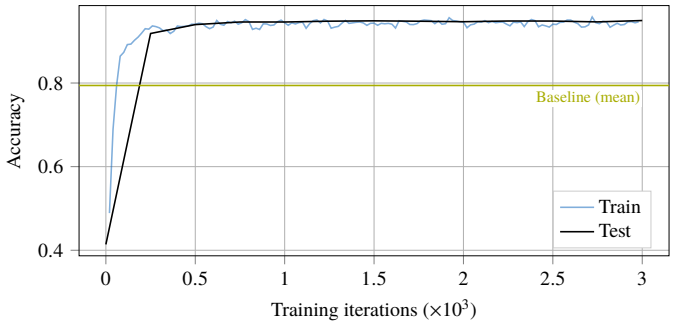

Figure 7: Training of *DeepMPLS* for trace generation using pre-trained weights.

The same approach of knowledge transfer was used for training *DeepMPLS* against the *Partial Synthesis* task, resulting in faster training convergence, with a training curve similar to the one illustrated in Figure 7.

D. Model Performance

We next assess the performance of *DeepMPLS* in the three different tasks presented in Section II-B.

1) Satisfiability task: We evaluate here the performance of *DeepMPLS* at predicting if a query is satisfiable or not. We use the prediction accuracy, precision and recall as metrics for evaluating the model.

Figures 8 and 9 illustrate the accuracies, precision and recall of *DeepMPLS* and the baseline. In average, *DeepMPLS* is able to predict the satisfiability of a query with an accuracy of $95.4\,\%$, a precision of $97.9\,\%$, and a recall of $89.2\,\%$. While the performance of the baseline drops with the size of the push-down automaton, *DeepMPLS* still performs well on those more complex cases.

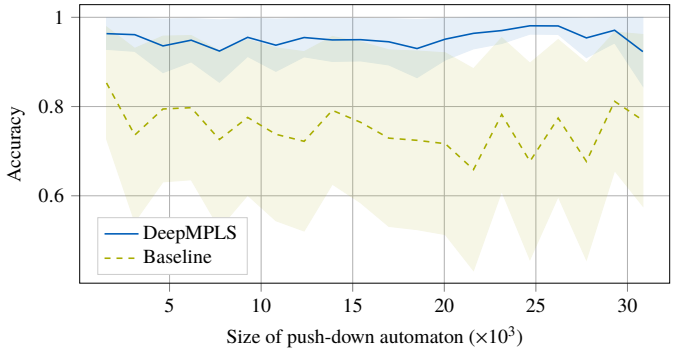

Figure 8: Accuracy of *DeepMPLS* and baseline against size of push-down automaton. Bands indicate the variance of the accuracy according to the push-down automaton sizes.

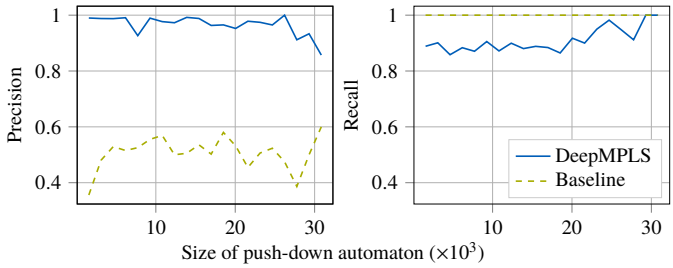

Figure 9: Precision and recall of *DeepMPLS* and baseline against size of push-down automaton.

2) Routing trace task: We evaluate here the performance of *DeepMPLS* at predicting the routing trace if a query is satisfiable. *DeepMPLS* was able to classify the routers with an overall accuracy of $92.8\,\%$ and a precision of $91.5\,\%$. If we redefine the accuracy as the correct prediction of all routers in a given topology, this per-topology accuracy of *DeepMPLS* is of $68.2\,\%$ in average.

Figure 10 illustrates this per-topology accuracy, with the detail of true positives and true negative. *DeepMPLS* has good performance for the true negatives with an average of $99.4\,\%$, while it reaches only a average of $85.6\,\%$ for the true positives. This means that routers in the true routing trace are sometimes missing in the prediction, but routers absent from the true routing trace are rarely selected (i.e. low false negative rate).

3) Partial synthesis task: Finally, we evaluate here the accuracy of for the *Partial synthesis* task, namely predicting

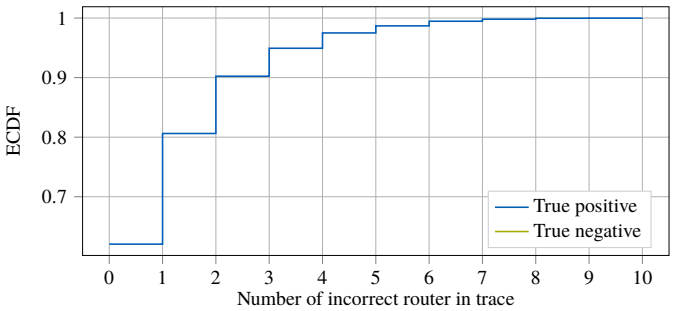

Figure 10: Per-topology accuracy of *DeepMPLS* with details on true and false positives.

which additional rules needs to be installed in a network in order to satisfy a query. Since the baseline described in Section IV-B cannot be applied here, we define a new one for this task. This new baseline randomly selects the requested number of edges in the *DeepMPLS* graph model following a simple random sampling without replacement.

Figure 11 illustrates the detailed per-topology accuracy of *DeepMPLS* and the baseline. In average, *DeepMPLS* is able to predict the correct edges with an accuracy of $45.9\,\%$, while the random baseline predicts them with an average accuracy of only $0.1\,\%$.

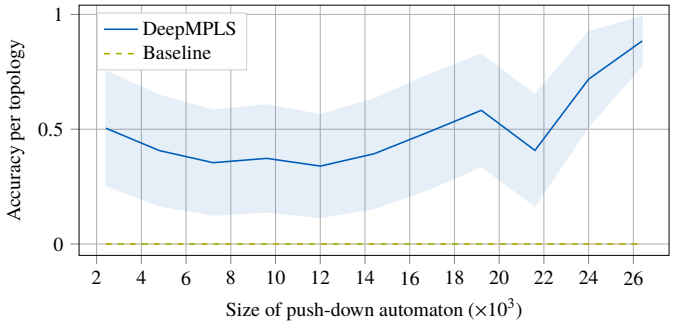

Figure 11: Per-topology accuracy of *DeepMPLS* and baseline.

E. Execution Time

In order to understand the practical applicability of *DeepMPLS*, we evaluate in this section its execution time in different settings. This part is a complement to the complexity analysis presented in Section II-C. We define and measure the execution time per network as the total time taken to process the network and a satisfiability query, without including the startup time or the time taken for initializing the network data structures.

Since the neural network can be evaluated on either CPU or GPU, we evaluated *DeepMPLS* on both platforms. A Nvidia GTX 1080 Ti was used for the measurements on GPU, and an Intel Xeon Gold 6130 CPU was used for the ones on CPU. The same CPU was used for the execution time measurement of P-Rex.

Figure 12 illustrates the different execution times and compares *DeepMPLS* against P-Rex. For the three different

evaluations, we note a linear relationship between size of the push-down automaton – and hence size of the analyzed graph – and the execution time. *DeepMPLS* is one order of magnitude faster than P-Rex when running on CPU, and two order of magnitudes faster on GPU, mainly due to the better ability of GPUs of parallelizing the numerical operations used in neural networks. Those figures illustrate that *DeepMPLS* show promising applicability due to fast computation times.

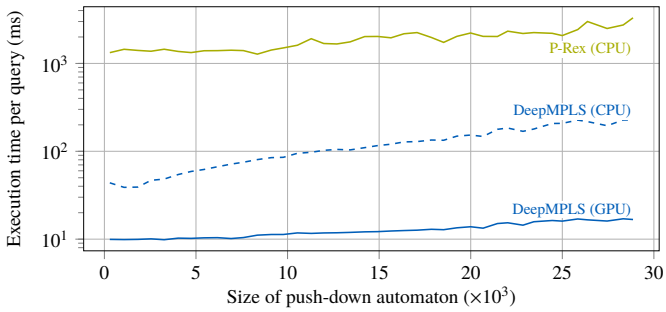

Figure 12: Execution time of *DeepMPLS* on CPU and GPU compared against P-Rex for the *Satisfiability* task.

F. Impact of Number of Iterations

We described in Section II-A that the GNN requires multiple evaluations of the recurrence defined in Equation (5) in order to propagate the hidden representations across multiple hops. We evaluate in this section the relationship between the number of iterations and the prediction accuracy of *DeepMPLS*.

Numerical results are illustrated in Figure 13 for the *Satisfiability* task. As the number of iterations increases, the prediction accuracy also increases. Convergence is reached after approximately 16 iterations. Since this parameter directly influences the execution time of *DeepMPLS*, a proper value has to be chosen in order to have a good trade-off between accuracy and computational complexity.

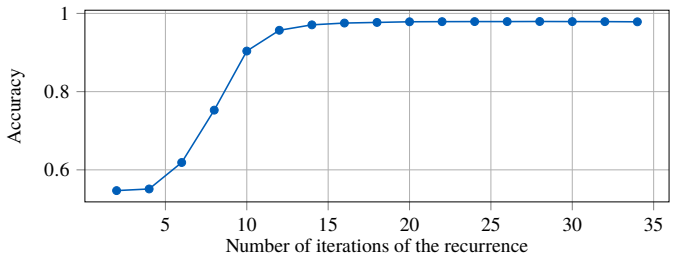

Figure 13: Impact of the number of iterations of Equation (5) on the prediction accuracy of the *Satisfiability* task.

V. Related Work

Motivated by the problems arising from the complexity of manual network operations especially under link failures [15, 16], much progress has been made over the last years towards more automated network operation and verification [2, 17, 18, 19, 3, 20, 21]. Existing network verification tools are typically fed with some model or configuration of the control plane and/or the data plane, and some query. While some tools are specific to a certain protocol, e.g., BGP [22], others are generic [3]. A well-known tool is NetKAT [2] which supports static verification of reachability, loop-freedom or waypoint enforcement, of the network configuration. Other well-known tools include HSA [3] (which is based on a geometric model from the packet headers ignoring protocol-specific meanings), VeriFlow [20] (acting as a layer between the network and an SDN controller), or Anteater [18] (based on a SAT solver).

In contrast to these works, we in this paper focus on MPLS networks, which are in wide use [23]. In particular, we are motivated by a recent line of research which showed that MPLS networks can be verified in polynomial-time, using a connection to prefix rewriting systems and automata theory [5]. The resulting tool, P-Rex [4], relies on a natural query language based on regular expressions which we also adopt in this paper. However, while efficient in theory and much faster than state-of-the-art tools in practice, especially under multiple link failures, P-Rex still requires an hour or more to test complex but relatively small networks.

We in this paper presented a first study of the feasibility of using deep learning to support faster answers to MPLS queries, as well as to synthesize configurations: an emerging topic [24, 25, 15, 26, 27, 28, 29] which to the best of our knowledge however has not yet been studied in the context of MPLS networks so far.

While our methodology is novel in this context, Graph Neural Networks have been around for quite some time [6, 7], and have also been extended to Gated Graph Neural Networks in [11], by using GRU memory units [12]. Message-passing neural networks were introduced in [9], with the goal of unifying various GNN and graph convolutional concepts. Veličković et al. [30] formalized graph attention networks, which enable to learn edge weights of a node neighborhood. Finally, [31] introduced the graph networks (GN) framework, a unified formalization of many concepts applied in GNNs. While existing applications are broad, including chemistry with molecule analysis [8, 9], jet physics and elementary particles [10], prediction of satisfiability of SAT problems [32], or basic logical reasoning tasks and program verification [11], only recently, first applications in networking have emerged, e.g., in the context of network calculus [33, 34], queuing theory [35], protocol generation [36], or the performance evaluation of networks with TCP flows for predicting average flow bandwidth [37].

VI. Conclusion

This paper showed that deep learning can not only be used for a faster verification of the policy-compliance of MPLS configurations, but even has the potential to provide efficient synthesis, automatically re-establishing certain network properties. To achieve this, *DeepMPLS* relies on a novel extension of graph-based neural networks. Our prototype implementation

shows promising results, in terms of runtime and accuracy, in realistic scenarios.

In general, we understand our paper as a first step and believe that our work opens several interesting directions for future work. In particular, we believe that our approach can be refined and optimized further, to provide an even better performance. Furthermore, it will be interesting to investigate the synthesis of full MPLS configurations based on reinforcement learning, or to test and generalize our approach for other configurations, e.g., based on Segment Routing.

In order to facilitate future research in this area and build upon our work, as well as to ensure reproducibility, we made the generated experimental data available online.

ACKNOWLEDGMENTS: The authors would like to thank Jiří Srba for his comments on a early version of this work. We are also grateful to our shepherd, Alex Liu, for his feedback and support. This work was supported by the German-French Academy for the Industry of the Future.

REFERENCES

[1] S. Peter, U. Javed, Q. Zhang, D. Woos, T. Anderson, and A. Krishnamurthy, "One tunnel is (often) enough," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 99–110.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *SIGPLAN Not.*, vol. 49, no. 1, Jan. 2014.

[3] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. of USENIX NSDI*, 2012.

[4] J. S. Jensen, T. B. Krogh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures," in *Proc. 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[5] S. Schmid and J. Srba, "Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks," in *Proc. of IEEE INFOCOM*, 2018.

[6] M. Gori, G. Monfardini, and F. Scarselli, "A New Model for Learning in Graph Domains," in *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, ser. IJCNN'05, vol. 2. IEEE, Aug. 2005, pp. 729–734.

[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.

[8] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Proc. of NIPS*, 2015.

[9] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. of NIPS*, 2017.

[10] I. Henrion, K. Cranmer, J. Bruna, K. Cho, J. Brehmer, G. Louppe, and G. Rochette, "Neural message passing for jet physics," in *Proc. of the Deep Learning for Physical Sciences Workshop*, 2017.

[11] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated Graph Sequence Neural Networks," in *Proc. of ICLR*, 2016.

[12] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. of EMNLP*, 2014.

[13] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 1, pp. 39–47, Mar. 1960.

[15] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations," in *Proc. of ACM SIGCOMM*. ACM, 2016, pp. 328–341.

[14] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet Topology Zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.

[16] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *Proc. of USENIX OSDI*, 2016.

[17] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A coalgebraic decision procedure for netkat," in *ACM SIGPLAN Notices*, vol. 50 (1), 2015, pp. 343–355.

[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM Computer Communication Review*, vol. 41 (4), 2011, pp. 290–301.

[19] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. ACM, 2017, pp. 155–168.

[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: verifying network-wide invariants in real time," in *Proc. of USENIX NSDI*, 2013, pp. 15–27.

[21] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. of USENIX NSDI*, 2014, pp. 87–99.

[22] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott, "FSR: Formal analysis and implementation toolkit for safe interdomain routing," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1814–1827, 2012.

[23] Y. Vanaubel, P. Mérindol, J.-J. Pansiot, and B. Donnet, "MPLS Under the Microscope: Revealing Actual Transit Path Diversity," in *Proc. ACM IMC*, 2015.

[24] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 261–281.

[25] ——, "NetComplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. of USENIX NSDI*, 2018.

[26] Propane Language. [Online]. Available: https://propane-lang.org/

[27] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Network Configuration Synthesis with Abstract Topologies," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. ACM, pp. 437–451.

[28] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Netcomplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. of USENIX NSDI*, 2018.

[29] ——, "Network-wide configuration synthesis," in *Proc. International Conference on Computer Aided Verification (CAV)*. Springer, 2017.

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. of ICLR*, 2018.

[31] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018, arxiv:1806.01261.

[32] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT Solver from Single-Bit Supervision," Feb. 2018.

[33] F. Geyer and G. Carle, "The Case for a Network Calculus Heuristic: Using Insights from Data for Tighter Bounds," in *Proc. of NetCal*, 2018.

[34] F. Geyer and S. Bondorf, "DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks," in *Proc. IEEE INFOCOM*, 2019.

[35] K. Rusek and P. Cholda, "Message-Passing Neural Networks Learn Little's Law," *IEEE Commun. Lett.*, 2018.

[36] F. Geyer and G. Carle, "Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning," in *Proc. SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks (Big-DAMA)*. ACM, Aug. 2018, pp. 40–45.

[37] F. Geyer, "Performance Evaluation of Network Topologies using Graph-Based Deep Learning," in *Proc. of EAI ValueTools*, 2017.