Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Prediction of TCP Performance Metrics Using Deep Graph Neural Networks

Lars Schwegmann

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Prediction of TCP Performance Metrics Using Deep Graph Neural Networks

Vorhersage von TCP Performanz Metriken unter Verwendung von Deep Graph Neural Networks

Author:	Lars Schwegmann
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Benedikt Jaeger, M. Sc.
	Max Helm, M. Sc.
Date:	May 15, 2022

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, May 15, 2022

Location, Date

Signature

Abstract

The prediction of Transmission Control Protocol (TCP) performance metrics can be a very powerful tool for the optimization of networks for certain characteristics such as the throughput of a certain TCP flow. However, such metrics are difficult to predict as multiple TCP flows in larger networks tend to behave chaotically. It has been shown that the recently emerged field of Gated Graph Neural Networks (GGNNs) can be successfully applied for the prediction of metrics in computer networks.

This thesis shows that GGNNs can be sucessfully utlized for the prediction of mean TCP flow Round-Trip Times (RTTs) and mean flow rates resulting in a best case mean absolute relative error of 1.66 % for RTT and 9.57 % for flow rate prediction against a test dataset. A model for predicting queue utilization is provided as well, resulting in a mean absolute error of 0.8 % for the best case scenario. Modeling individual interfaces instead of only modeling network nodes in the graph representations is shown to improve model accuracy by an order of magnitude. Graph representations with additional path ordering nodes are found to improve model performance for queue utilization and flow rate predictions, while not making much of a difference in RTT predictions. The generalization of the models to larger network sizes is found to be somewhat limited in accuracy, as significantly longer path lengths compared to the training data cause larger errors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The ability to predict Transmission Control Protocol (TCP) flow performance metrics, such as throughput, latency, and loss, can be very useful for optimizing networks for certain goals. Due to many factors which can influence a flows performance, TCP flows tend to act almost chaotically and these metrics become very hard to predict once multiple flows are involved. Factors which might influence a TCP flow usually include the layout of the network topology and its links, the TCP congestion control strategies which are in use, the other flows present on the network and the queues at the hosts which the flow traverses.

There have been attempts to formalize the behaviour of certain TCP algorithms. The most notable example is probably the approach by Padhye et al. [1], in which they model the TCP Reno steady-state send rate as a function of loss rate and Round-Trip Time (RTT). However, these analytical approaches fail to model complete networks with multiple flows and congestion strategies interacting with each other.

In order to obtain predictions with a high enough accuracy for a whole network, one must usually resort to discrete event network simulators or actually building a testing setup with real hardware. Both of these approaches do not allow for changing things and reiterating quickly though. Because discrete event netwok simulators such as *ns3* [2] or *OMNet++* [3] actually simulate each packet passing through the network the process usually cannot be parallelized. As such, discrete event network simulation becomes inviable for optimization purposes once the networks to be simulated reach a certain size or the flows reach a certain throughput (since higher throughput usually means more

packets). As shown by Geyer [4], it is possible to acquire sufficiently accurate prediction results with significantly less computational overhead per iteration using methods from the fairly recent field of Gated Graph Neural Networks (GGNNs).

1.2 Research questions

In the following subsections, research questions are proposed which this thesis aims to find an answer to.

1.2.1 Effects of different graph representations

The first research question concerns how different ways to represent the network topology in the Machine Learning (ML) model affect the model's accuracy. The network topology layout and the information about the flows, queues, etc. have to be mapped to a graph in order for a Graph Neural Network (GNN) to be able to process the information. There are many different ways to represent these networks, and a central goal of this thesis is to find out which kind of representations work well and which do not. Detailed information on how these graph representations are created can be found in Section 4.3.

1.2.2 Ability to generalize to larger topology sizes

Another interesting aspect is the accuracy of the ML model when it is confronted with data it has not encountered before. Especially interesting for the problem of TCP performance prediction is the ability of the model to scale to larger topology sizes it was not trained with, as larger topologies would take longer to simulate in a traditional discrete event network simulator. The generalization in regards to other properties of the network is also interesting to consider. For example, it could be tested how well a ML model adapts to changes in queue sizes.

1.2.3 Effects of feature choice on model performance

Finally, feature importance is a very interesting topic to investigate. Concretely, the goal is to find out which features affect the model's accuracy and if so, by what amount. Another aspect would be finding out which features are most important depending on the predicted metric. For example, does including the queue size in the model's features make the same difference for both bandwidth and RTT prediction?

Chapter 2

Background

In this chapter, we give a general overview over the basics of Machine Learning as well as an introduction to Gated Graph Neural Networks and their applications for predicting metrics in computer networks.

2.1 Machine Learning fundamentals

ML describes the general process of "learning" a function from data. This is useful for tasks where it would be tedious or simply impossible to write a program for solving said tasks, because no well-defined algorithm for solving the task exists. Prominent examples for such machine learning use-cases include image classification, speech recognition and object detection tasks. Instead of implementing a program which solves the task directly, one implements a so-called learning algorithm. The learning algorithm *trains* the ML model using a training dataset of example data points. A training example consists of *features*. Depending on the task, these datasets are collected or generated beforehand [5].

ML algorithms can be divided into three broad categories: supervised, unsupervised and reinforcement learning algorithms. Unsupervised learning algorithms are only exposed to the dataset itself with the intent to learn information about the structure and probability distribution of the dataset. Supervised learning algorithms on the other hand are given an expected result for each example data point called *label* and are expected to learn the mapping from the dataset to the given labels. Reinforcement learning algorithms differ from both supervised and unsupervised learning algorithms, as they try to maximize a reward function instead of trying to find the structure of the dataset, but are not given any expected results like in supervised learning algorithms [5][6].

In the following sections, we will focus on supervised learning methods as that is what we require for the models presented later in this thesis. We will also focus only on so-called *regression* tasks, which means that we want an ML model to predict numerical values, or in other words, have the learning algorithm learn a function $f : \mathbb{R}^n \to \mathbb{R}^m$ which maps the input features to a vector of numerical output values [5].

In supervised learning algorithms, the accuracy of the model is usually judged using an error function based on the expected and actual output of the model. This function is sometimes referred to as *criterion*. The criterion can be chosen freely, and thus, can be considered part of the model's *hyperparameters*. Hyperparameters influence the learning process, but are not part of the function the model learns. Instead, they can be thought of as settings, which influence how the model learns [5].

2.2 Graph Neural Networks

For many applications, including the field of computer networks, it makes sense to represent data as graph structures. However, due to the architecture of traditional neural networks, the graph cannot be fed into a neural network as is. Instead, it needs to be transformed into a simpler matrix representation. This preprocessing usually does not preserve the relationships of nodes in the graph structure. Since this relationship information is essentially lost, these ML models usually do not generalize well to other topology layouts [7][8].

GNNs were first proposed by Gori et al. [9] as an extension to Recurrent Neural Networks (RNNs) in 2005 in order to get rid of these preprocessing steps. They propose a solution which encodes the structure of the input graph in the neural network itself. The more recent work by Scarselli et al. [7] goes into more detail and is used as primary reference material for this section.

In the following, undirected graphs $G = (\mathcal{N}, \mathcal{E})$ with nodes $n \in \mathcal{N}$ and edges $e \in \mathcal{E}$ are considered. Please note that the GNN architecture, as proposed by Scarselli et al. [7], is also applicable to directed graphs, but is not considered here for simplicity reasons. Edges can be denoted by the pair of nodes they are connected to, as in $e = (n, n')$. Nodes and edges may also hold information (features) in the form of labels denoted by $l_n \in \mathcal{L}$ and $l_{(n,n')} \in \mathcal{L}$. The set of direct neighbouring nodes of nodes $n \in \mathcal{N}$ is denoted by $ne[n]$. The set of edges containing nodes $n \in \mathcal{N}$ is denoted by $co[n]$.

Graphs are typically used to model relationships between certain entities. In order to model these connections in the neural network, Scarselli et al. [7] propose that each node n should maintain a state h_n which is influenced by the neighbouring nodes' and edges'

states and features. This influence is modeled by the *local transition function* f_w which is used to combine the neighbouring states and features to compute the local state h_n [7].

$$h_n = f_w(l_n, l_{ne[n]}, h_{ne[n]}) \tag{2.1}$$

This state propagation, also called *neural message passing* in the context of GNNs, is done iteratively until a fixed point is reached. Convergence is assured if f_w is chosen so that the global transitioning function F_w of the entire network is a contraction map. This is possible because of Banach's fixed point theorem, which states that a unique solution always exists and that F_w will always converge to that fixpoint, no matter the given input values [7].

The output vector o_n of each node can be calculated after reaching this fixed point using the *local output function* g_w, which takes the state h_n and the features l_n of node n as inputs.

$$o_n = g_w(h_n, l_n) \tag{2.2}$$

Scarselli et al. [7] suggest that, in practice, it makes sense to reformulate Equation 2.1 as a sum of the terms for each incoming edge as follows:

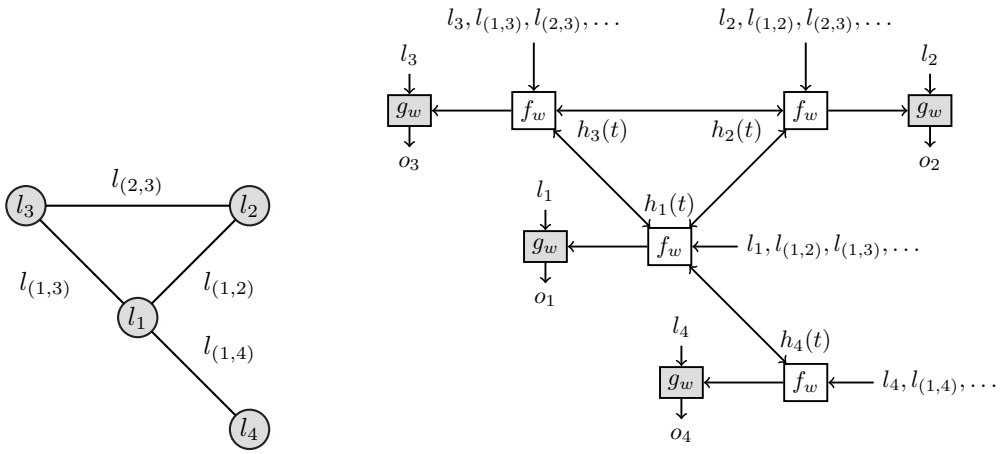
$$h_n = \sum_{u \in ne[n]} f_w(l_n, l_{(n,u)}, h_u, l_u) \tag{2.3}$$

The function f_w can be implemented either by a simple linear relationship or by a Feed-Forward Neural Network (FFNN). g_w is usually implemented as a multilayered FFNN. Both functions are parameterized with weights and biases, which can be learned by training the network [7].

Using the original graph structure, an RNN is built using f_w and the labels/features of the input graph. As shown, for example, in Figure 2.1, the graph's nodes are replaced by units implementing f_w and g_w. These units store the current state of their respective node $h_n(t)$ for the current timestep t and calculate $h_n(t+1)$ when activated. The resulting RNN is called *encoding network* [7]. The relationship between the state of a node and its previous state can be formulated as follows:

$$h_n(t+1) = f_w(l_n, l_{co[n]}, h_{ne[n]}(t), l_{ne[n]}) \tag{2.4}$$

This encoding network in the form of an RNN can be *unrolled* to a FFNN where the same RNN forms a sequence and each instance represents another message passing



(a) Example graph

(b) Example encoding network

FIGURE 2.1: Example graph and a corresponding encoding network. Figure adapted from Scarselli et al. [7]

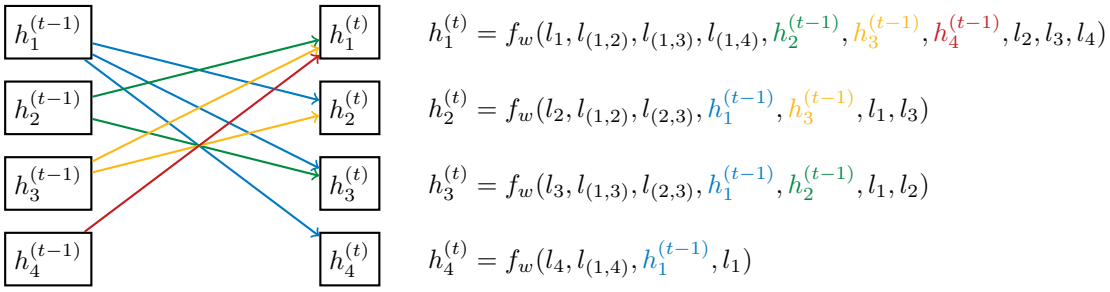


FIGURE 2.2: Visualization of a single iteration of state propagation for the graph from Figure 2.1a. Figure adapted from Geyer [4]

timestep. Figure 2.2 shows how this message passing works for the given example graph from Figure 2.1a for a single unrolled timestep. The state vector $h_n^{(t)}$ is usually referred to as *hidden state* in RNNs [7][4].

Scarselli et al. [7] show that the resulting RNN is differentiable from end to end, and thus the network can be trained using a standard gradient-descent based training strategy. For this, the node states h_n are updated iteratively until a fixed point is reached. Then, a gradient can be computed and the weights can be updated by backpropagation. This is done using the Almeida-Pineda algorithm for backpropagation in RNNs, as outlined by Scarselli et al. The necessity for the contraction map property of F_w may limit the expressiveness of the model [10].

2.3 Gated Graph Neural Networks

Gated Graph Neural Networks (GGNNs) extend the framework of GNNs laid out by Gori et al. [9] and Scarselli et al. [7]. They were first introduced by Li et al. [10] and provide ways for the neurons in the network to have internal memory, essentially allowing the neurons to selectively hold on to their state from previous iterations. This is achieved using previous work by Cho et al. [11] on so-called Gated Recurrent Unit (GRU) cells.

2.3.1 Gated Recurrent Units

GRUs were proposed in order to allow RNNs to selectively remember and forget information about sequences. The following subsection is a summary of the work of Cho et al. [11] with the terms adjusted to GGNNs by Li et al. [10].

In the context of GNNs, you can think of the GRUs to replace the local transitioning function f_w introduced in Section 2.2. The GRU cell of each node in the encoding network processes the input from the surrounding nodes $a_n^{(t)}$ and the hidden state of the previous iteration $h_n^{(t-1)}$ of the node and computes the new hidden state $h_n^{(t)}$ of the current iteration. Note that one could still add pre- or postprocessing functions in the form of additional neurons before or after the GRU respectively. This is not considered here for the sake of simplicity though.

Figure 2.3 shows the inner workings of a GRU cell as a flow diagram. The inputs (previous hidden state $h_n^{(t-1)}$, input state $a_n^{(t)}$) are shown on the left and the output (new hidden state $h_n^{(t)}$) is shown on the right.

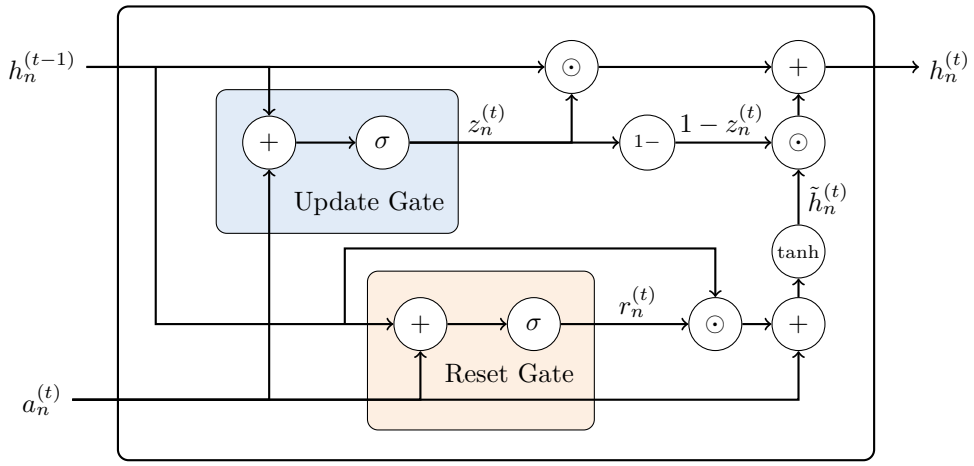


Figure 2.3: Inner structure of a GRU cell

The basic propagation model works as follows: First, A is defined to be the adjacency matrix of the graph, and A_n to be the adjacency column vector corresponding to node $n \in \mathcal{N}$. With this, the input state vector of the GRU cell can be defined as

$$a_n^{(t)} = A_n \left[h_1^{(t-1)} \dots h_{|\mathcal{N}|}^{(t-1)} \right]^\top + b_a \tag{2.5}$$

which is essentially a vector that combines all the neighbouring hidden state vectors from the previous timestep and adds an additional bias b_a. The GRU consists of two main logical function blocks called *reset gate* $r_n^{(t)}$ and *update gate* $z_n^{(t)}$. They are implemented as follows:

$$r_n^{(t)} = \sigma(W_r a_n^{(t)} + U_r h_n^{(t-1)} + b_r) \tag{2.6}$$

$$z_n^{(t)} = \sigma(W_z a_n^{(t)} + U_z h_n^{(t-1)} + b_z) \tag{2.7}$$

Here, W_r, W_z, U_r and U_z are learnable weight matrices, b_r and b_z are learnable biases and σ is the logistic sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ which is used to fix the range of results to $[0, 1]$. Intuitively, the update gate output vector determines how much information of the previous iteration is passed along into the state of the current iteration. The reset gate output vector determines what and how much information gets discarded from the previous hidden state. The actual discarding is done in the following calculation for the intermediate hidden state $\tilde{h}_n^{(t)}$

$$\tilde{h}_n^{(t)} = \tanh(W a_n^{(t)} + U(r_n^{(t)} \odot h_n^{(t-1)}) + b) \tag{2.8}$$

with W and U being more learnable weight matrices, b being another learnable bias and $\odot$ being the Hadamard product (element-wise matrix multiplication). If an element in $r_n^{(t)}$ is learned to be close to zero, the respective element in $h_n^{(t-1)}$ is "forgotten". The tanh function is used on the resulting vector to fix the values to the interval $[-1, 1]$ [10]. $\tilde{h}_n^{(t)}$ is then fed into the following equation for the final output hidden state $h_n^{(t)}$

$$h_n^{(t)} = (1 - z_n^{(t)}) \odot h_n^{(t-1)} + z_n^{(t)} \odot \tilde{h}_n^{(t)} \tag{2.9}$$

Here, the update gate output $z_n^{(t)}$ determines what information of the intermediate hidden state $\tilde{h}_n^{(t)}$ and the previous hidden state $h_n^{(t-1)}$ is passed along to the output $h_n^{(t)}$. The output vector o_n of the node can be calculated after the RNN iterations using Equation 2.2 [10].

For the first iteration, the nodes' hidden state vectors $h_n^{(0)}$ are intialized to contain values from their respective node label vectors l_n. Since one might want the hidden state vector to be larger than the label vectors, the remaining rows are padded with zeros:

$$h_n^{(0)} = \left[l_v^\top,\ 0 \right]^\top \tag{2.10}$$

or the input features are passed through a FFNN with an output size equal to that of the hidden state vector [10][4].

2.3.2 Long Short-Term Memory Cells

Long Short-Term Memory (LSTM) cells are a slightly different approach for memory cells in neural networks. They were fist proposed by Hochreiter et al. in 1997 [12]. In GGNNs, they can replace the GRU memory cell introduced in Section 2.3.1. The following is a summary of the work by Hochreiter et al. [12], with the terms adjusted for GGNN by Geyer [4].

Compared to GRU cells, LSTM cells are a bit more complex, as they have an additional gate called output gate. However, Geyer [4] has found that LSTM cells can produce better results in certain situation when modeling computer networks.

The inner strucure of an LSTM cell is shown in Figure 2.4. On the left side, the cell takes the previous cell state $c_n^{(t-1)}$, the previous hidden state $h_n^{(t-1)}$ and the input from the surrounding nodes $a_n^{(t)}$ as inputs. The outputs, namely the new cell state $c_n^{(t)}$ and the new hidden state $h_n^{(t)}$, are shown on the right side of the cell.

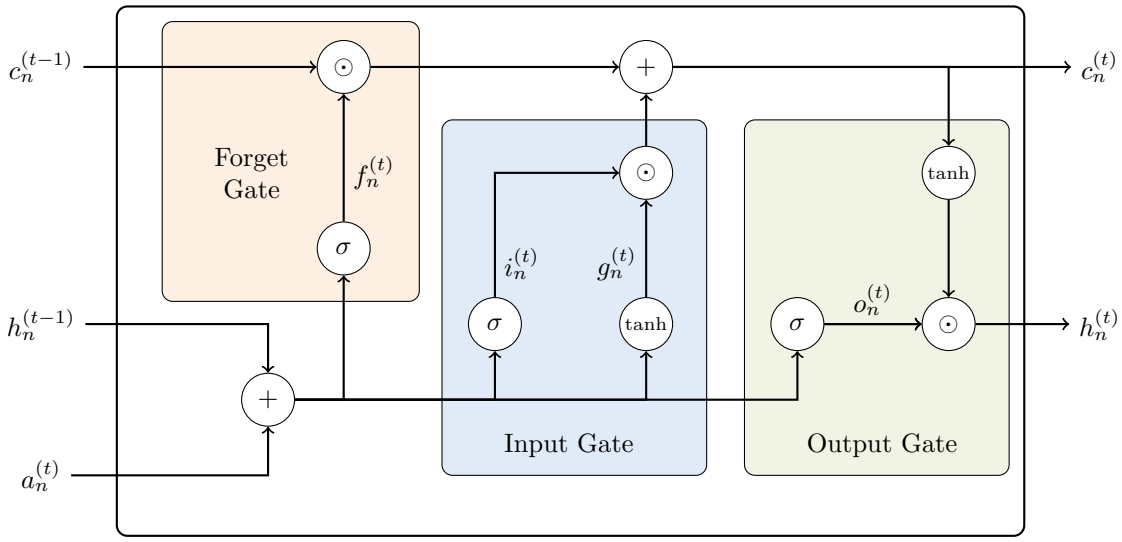


Figure 2.4: Inner structure of an LSTM cell

Due to a different inner structure, the propagation model differes quite a bit compared to the GRU cell. Again, A is defined to be the adjacency matrix of the graph, and A_n is defined to be the column vector from A corresponding to node $n \in \mathcal{N}$. The input state for timestep t can be defined as

$$a_n^{(t)} = A_n \left[h_1^{(t-1)} \dots h_{|\mathcal{N}|}^{(t-1)} \right]^\top + b_a \tag{2.11}$$

The LSTM cell consists of three function blocks, the *input gate* $i_n^{(t)}$, the *forget gate* $f_n^{(t)}$, and the *output gate* $o_n^{(t)}$. The forget gate decides what information from the previous cell state $c_n^{(t-1)}$ is "forgotten". It is implemented using the following equation:

$$f_n^{(t)} = \sigma(W_f a_n^{(t)} + U_f h_n^{(t-1)} + b_f) \tag{2.12}$$

Here, W_f and U_f are learnable weights and b_f is a learnable bias. As shown in the diagram, the vector $f_n^{(t)}$ is multiplied element-wise with the previous cell state $c_n^{(t)}$. A value close to zero in $f_n^{(t)}$ leads to the value in $c_n^{(t)}$ to be discarded [4].

Next is the input gate, shown in the blue box in the diagram. First, the vector $i_n^{(t)}$ is computed, using the following, similiarly looking equation:

$$i_n^{(t)} = \sigma(W_i a_n^{(t)} + U_i h_n^{(t-1)} + b_i) \tag{2.13}$$

with weights W_i and U_i and bias b_i. This vector is used to determine which values of the input are going to be used to update the new cell state. They are multiplied element-wise with the vector

$$g_n^{(t)} = \tanh(W_g a_n^{(t)} + U_g h_n^{(t-1)} + b_g) \tag{2.14}$$

which can be viewed as an update candidate vector calculated from the previous hidden state and cell input. This equation is also weighted with weights W_g and U_g and biased with b_g.

The new cell state $c_n^{(t)}$ is obtained by adding the previous cell state (with some values already discarded by the forget gate) to the output of the input gate:

$$c_n^{(t)} = c_n^{(t-1)} \odot f_n^{(t)} + g_n^{(t)} \odot i_n^{(t)} \tag{2.15}$$

Finally, the output gate transforms the input and previous hidden state once more using yet another set of weights W_o, U_o and bias b_o

$$o_n^{(t)} = \tanh(W_o a_n^{(t)} + U_o h_n^{(t-1)} + b_o) \tag{2.16}$$

to obtain $o_n^{(t)}$, which determines which values from $c_n^{(t)}$ are output to the new hidden state $h_n^{(t)}$. For this, $o_n^{(t)}$ is multiplied element-wise with the tanh of $c_n^{(t)}$:

$$h_n^{(t)} = \tanh(c_n^{(t)}) \odot o_n^{(t)} \tag{2.17}$$

In summary, LSTM cells differ from GRU cells in their gating logic. They also store an additional cell state vector $c_n^{(t)}$ in between iterations, making them a bit more complicated to implement. Although differently implemented, both GRU and LSTM cells both aim to solve the same problem, namely helping the network to selectively remember or forget information passing through the nodes.

2.3.3 Message passing and learning in GGNNs

In order to calculate the output vector o_n for each node n, a fixed amount of message passing iterations is done instead of iterating until a fixed point is reached. This allows the use of gradient-based training methods commonly used in FFNN. The number of iterations to unroll affects the model's accuracy and can be chosen freely. However, one has to keep in mind that an unroll count which is too small in relation to the input graph will result in poor accuracy, while an unroll count which is too high will result in poor training and inference performance [4].

2.4 GGNNs for computer networks

The previously introduced GGNNs can be applied to learn arbitrary functions on graphs. As stated in Chapter 1, this thesis' aim lies in the prediction of the performance of TCP flows in arbitrary computer networks. Problems on computer networks can be modeled as graph problems, as was shown by Geyer [4]. The following is largely based on their previous work.

For the ML model, the network topology is transformed into a graph which not only models the original structure of the topology, but also adds additional nodes and edges specific to the modeled problem. We call these transformed versions of the original network graphs *Graph Representations (GRs)* [4].

To illustrate how such a transformation works, consider the example network topology shown in Figure 2.5. Here, a computer network consisting of three routers, six servers and three unidirectional flows is shown. Note that the flows all start and end at servers. In addition to the topology, information about attributes of the links and flows between nodes in the network as shown in Table 2.1 is known. Using this information, a possible GR can be constructed, as depicted in Figure 2.6a. This is only one of many possible ways to represent the input computer network and the accompanying attributes. This specific representation only models the sending network interfaces of the servers and routers, which the flows are passing through. The interface nodes hold information about the links they are connected to, such as bandwidth and delay. They are in turn connected to flow nodes, which hold the measured flow information, such as the flow rate.

Note that there are two differing node types in this GR, flow nodes and interface nodes. One could add more node types to the graph if it is deemed useful for the desired GR. The nodes from the computer network which no flow traverses have been omitted, e.g.

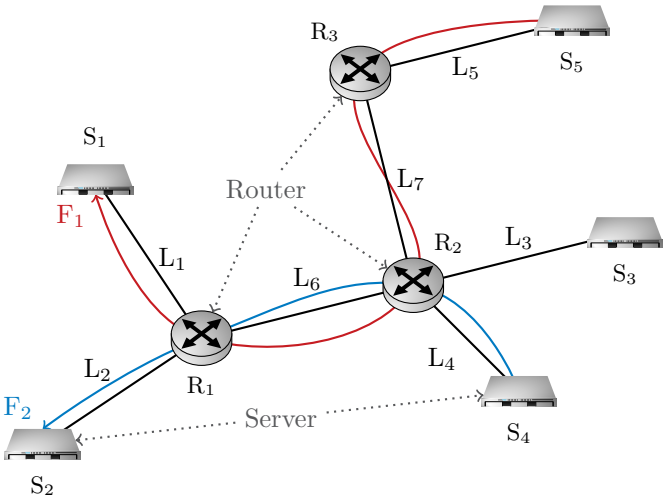


Figure 2.5: Example network topology

the sending interface node for S_3, as they would not be connected to any flow node anyways.

This GR can then be used for constructing the actual GGNN. For this task, the nodes' feature vectors l_n are all constructed to be the same length. The node type can be one-hot encoded, in our case using only two columns since we only have two possible node types. In Figure 2.6b, the corresponding node feature vectors are shown as a matrix, with each row corresponding to one node in the GR. In this example, we encode flow nodes with node type $[1\ 0]^\top$ and interface nodes with node type $[0\ 1]^\top$. The following entries in each row are all encoded features of said nodes. More specifically, the entries can be read from left to right as node type, flow rate, flow RTT, link bandwidth and link delay. The first two rows correspond to the two flow nodes F_1 and F_2 and all following nodes are the interface nodes ordered by their associated link node number. For example, row three shows node I_{R1S1} from the GR.

Each node feature vector has the same size, while the specific node type determines which features in the node have meaning. Note that all of the entries in the matrix are scaled so that the values all have the same order of magnitude. This can be done using min-max normalization for example, as was done here:

$$x \in S, \text{minmax}(x, S) = \frac{x - \min(S)}{\max(S) - \min(S)} \tag{2.18}$$

Using minmax normalization results in all values being mapped to a number between zero and one. Other normalization functions could be used as well, e.g. $\log(1 + x)$.

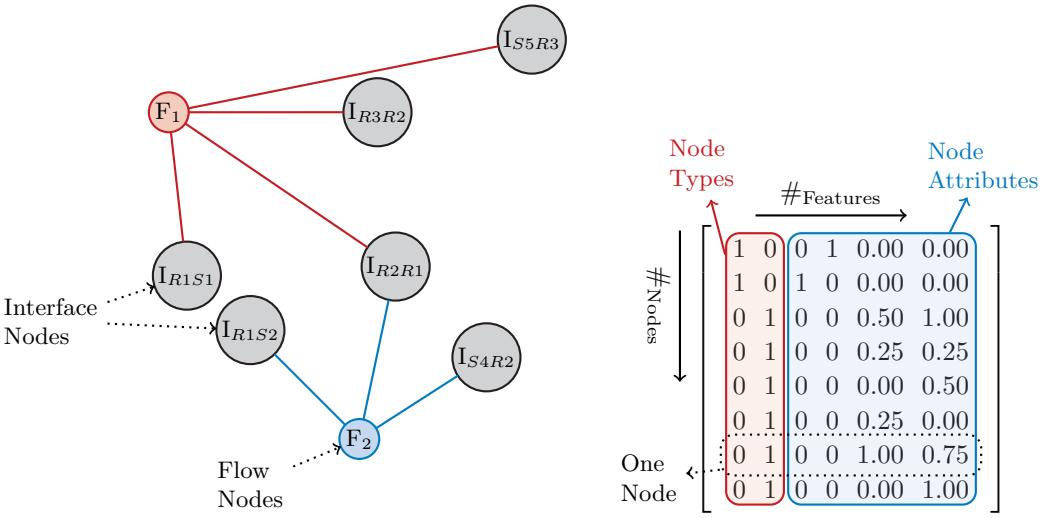
Flow	Flow Rate	RTT
F_1	10 Mbit/s	30 ms
F_2	20 Mbit/s	18 ms

(a) Measured flow attributes

Link	Bandwidth	Delay
L_1	30 Mbit/s	5 ms
L_2	20 Mbit/s	2 ms
L_3	50 Mbit/s	5 ms
L_4	10 Mbit/s	3 ms
L_5	20 Mbit/s	1 ms
L_6	50 Mbit/s	4 ms
L_7	10 Mbit/s	5 ms

(b) Link attributes

TABLE 2.1: Attributes of the example network topology from Figure 2.5



(a) Example network Graph Representation (b) Example Graph Representation matrix form

FIGURE 2.6: Example Graph Representation and matrix form

Using this or any other Graph Representation, a GGNN can be constructed as explained in Section 2.3.

Chapter 3

Related work

Over the years, many different approaches for the prediction of TCP flows have been proposed. In this chapter, we will give a brief overview of a few selected approaches which are relevant to this thesis.

3.1 SVR approach by Mirza et al.

The oldest of the listed related approaches was proposed by Mirza et al. in 2010 [13]. They apply a Support Vector Regression (SVR) ML algorithm to predict the throughput of TCP flows and compare the results against history-based TCP performance estimation. However, they do not approach this problem with the goal of being able to optimize a complete network topology, but rather from a perspective of selecting the best possible path for a flow in a fixed network. This approach yields flow throughput predictions, which are accurate to 10 % of the actual value 87 % of the time for bulk TCP transfers (steadily sending as much data as possible) [13]. These bulk TCP transfers are of special interest in regards to this thesis.

As stated, this approach uses SVR instead of GNNs for learning the TCP performance metrics. Due to this, the metrics Mirza et al. take into account for training are limited to available bandwidth for the flow, queueing delays and packet loss and do not consider the topology layout directly. They obtain their training data using a physical experiment setup similiar to a dumbbell topology, with artificially generated background traffic and delay emulation. In this topology, they do both active and passive measurements of said metrics. Active measurements are done before measuring the actual flow throughput using specialized software while passive measurements are conducted during the throughput

measurement. The passive measurements are done at the bottleneck link using detailed packet traces [13].

The previously stated accuracy is reached using oracular passive measurements (passive measurements of the complete path) of queueing delays and represents the best case scenario. They show that having the available bandwidth in the feature vector does not alter the accuracy in a meaningful way when using this oracular path measurement. Practical passive and active path measurements in the feature vector yield worse results, only predicting the throughput within a 10 % error margin 53 % and 51 % of the time, respectively [13].

3.2 Delay modeling by Mestres et al.

Mestres et al. [14] approach the problem of delay modeling in computer networks using ML. They focus on the optimization of networks for certain parameters, just like this thesis. However, they only explore the general viability of ML for this purpose and use traditional ML algorithms, namely FFNNs. Only three topology layouts are considered: A unidirectional ring, a star and a scale-free topology layout. Their training data is generated by *OmNET++*, a discrete event network simulator. The models are always trained on a single topology layout, as the layout cannot be fed into the network like in a GNN. The only feature the model receives as input is a traffic matrix with ingress and egress traffic for each node in the network, while the output vector is traind to contain the average end-to-end delay for each flow. Unfortunately, concrete measurements or accuracies with test datasets are not provided by Mestres et al. Only the learning error for the different topologies is given, which is not useful for talking about generalization of the model [14].

3.3 DeepComNet: GGNN approach by Geyer

Geyer [4] proposes a framework for network performance evaluation using deep GGNNs. They introduced the approach forming the basis of this thesis' ML model. The GGNN model and computer network graph representation is described in detail in Section 2.4. In the paper, Geyer introduces the concept of GGNN for computer networks and applies it to two use cases: Prediction of TCP flow bandwidths and prediction of User Datagram Protocol (UDP) flow end-to-end latencies. They implement three different GNN models: a GGNN using GRU cells, a GGNN using LSTM cells and a GNN with a simple RNN architecture. The evaluation of their TCP performance prediction is most interesting in regards to this thesis. For this, they randomly generated network topologies

of daisy-chained ethernet switches, then attached a random number of nodes to these switches and added flows with random node pairs to the network. The networks were simulated using *ns-2* (predecessor of ns-3 [2]). The graph representation was modeled like the following: Each TCP flow is encoded as two nodes, one for the data flowing from source to destination, and one for the TCP ACK segments flowing from the destination back to the source. These nodes are then connected to queue nodes modeling the queues of the nodes on the flow's path [4].

The results are then compared to SVR (comparable to approach by Mirza et al. [13] from Section 3.1) and FFNN approaches using high level input features. In the TCP evaluation, then GGNN-LSTM model perfoms best with a median relative error below 1 % [4]. In the TCP use-case, the GNN approaches all outperform both the SVR and FFNN approaches [4]. Geyer also shows that GGNN with GRU or LSTM memory cells positively impact the prediction performance. They find that an increase in the graph size increases the Mean Percentage Error (MPE) in the TCP use case, and that a larger number of unrolled loops in the GGNN yielded a lower relative error. In their evaluation, the effect has diminishing returns for more than 12 unrolled loops, as more unrolled loops lead to longer training and inference times [4].

3.4 RouteNet: GGNN approach by Rusek et al.

The *RouteNet* paper, published in 2020 by Rusek et al. [8], poposes a GNN approach to predict Key Performance Indicators (KPIs) of arbitrary networks, such as delay, jitter, and packet loss. It is an improved version of their previous implementation [15] and aims to provide an accurate but lightweight alternative to computational network simulators. Compared to other approaches like that of Geyer, RouteNet represents paths in the network as ordered sequences of links. The computer network is modeled only by these two sets of links and paths, where the link states are depending on the state of all paths traversing the link, and vice versa. In order to resolve this circular dependency, they rely on the repeated message passing in order to reach a fixpoint. They show, that when trained with various topologies ranging from 12 nodes to 50 nodes along with sample data obtained using OmNET++, the model yields a high prediction accuracy (MPE=15.4 % in the worst case) with data it was not trained with. Rusek et al. also provide a use case example by implementing a network optimizer using RouteNet for Software-defined Networking (SDN), which uses the KPI predictions from RouteNet to minimize the mean end-to-end delay of paths [8].

3.5 GGNN RouteNet-Erlang by Galmés et al.

RouteNet-Erlang (RouteNet-E) by Galmés et al. [16] can be viewed as a further improvement of the RouteNet paper introduced in Section 3.4. The overall premise and goal of RouteNet-E are still in line with those of RouteNet, but the model architecture was altered. The network is modeled as a set of links, queues and flows. The model follows three priciples: The state of flows is affected by the queues and links it passes over, the state of queues is affected by the flows passing over them, and the state of links is affected by the state of the queues at the output of the link and the queueing scheduling policy. Using this model, the network topology is transformed into the RouteNet-E input graph with the three given node types. Just like in RouteNet, these circular dependencies are resolved using multiple message passing iterations, although in this case, each iteration has three steps instead of two due to the additional dependency [8][16].

RouteNet-E is also trained using data obtained from the network simulator OmNET++. Galmés et al. evaluate RouteNet-E under two main aspects: The models ability to accurately predict the networks KPIs with different traffic models and the models accuracy under various different queueing strategies. Both times, the model output is compared to the results of a Queueing Theory (QT) approach. As expected, RouteNet-E outperforms the QT approach in both evalutations. In the worst case for RouteNet-E in the traffic model evaluation, the autocorellated exponential traffic model, RouteNet-E still achieves a mean absolute relative error of 11.95 % in jitter prediction, compared to 74.38 % with the QT approach [16].

Chapter 4

Approach

The following chapter outlines the approach to answer the research questions proposed in Section 1.2.

4.1 Overview of the training pipeline

Before training any ML model, one first needs data that the model can be trained with. It was decided to go with a discrete event network simulator, namely the *ns-3* [2] simulator instead of building actual network topologies and measuring real world data. This decision was made for the following reasons: Firstly, the results produced by ns-3 are reproducible and unaffected by potential disturbing factors present in actual hardware testbeds. Secondly, ns-3 does not constrain the topologies in terms of their size (only the compute resources and time do) whereas a physical testbed would. Finally, the data generation can be parallelized by running multiple simulations in parallel and does not require manual recabling of hardware, greatly increasing the speed at which the training data can be generated.

The results of these simulations can then be passed through a parser, which takes the simulated network topology and simulation results and transforms them into a graph representation with additional nodes to model the TCP flows. There are multiple ways to model these representations. These are discussed in more detail in Section 4.3.

The graph representation can then be used to train a GGNN. For this, a framework developed at the chair for training GGNNs in combination with the popular *PyTorch* framework is used. After training, the model can be evaluated for accuracy using a dataset unknown to the model.

For the implementation details of the training pipeline, please refer to Chapter 5.

4.2 Prediction metrics

It was decided to focus on the prediction of mean TCP flow rates, mean TCP RTTs and mean queue utilizations over the duration of the entire simulation, since these metrics are relatively easy to obtain using the simulator. However, the training pipeline and framework utilized in this thesis should allow for other metrics to be explored, such as loss for example.

4.3 Graph representations

In order to compare the performance of the ML model with different GRs, a selection of different meaningful representations was made. Due to a limitation in the ML model implementation, the GRs only ever include a single predicted feature (see Section 5.3 for details) at a time. However, they could of course be constructed to hold multiple predicted features in theory. They are introduced in the following subsections.

4.3.1 Graph representation 1

The first GR's basic structure was already introduced in Section 2.4. For this representation, the sending interfaces of each node in the network is modeled along with a single flow node for each flow. The flow nodes have connecting edges to all sending interfaces which the flow passes over. They have the used TCP congestion control algorithm and, if necessary, the predicted attribute (mean flow rate or mean flow RTT) as features. The interface nodes hold information about the link (link bandwidth, link delay) and the maximum queue size of the sending interface. They also hold the mean queue utilization, if predicted. For a visual representation, see Figure 2.6a.

4.3.2 Graph representation 2

The second GR is built on the same idea as GR 1, but adds dedicated path nodes between the flow nodes and the interface nodes. These path nodes hold an integer representing the index of the connected interface node in the route of the flow, thus modeling the flow's direction. See Figure 4.1 for a visual representation of the transformed example topology from Figure 2.5.

4.3.3 Graph representation 3

The third Graph Representation takes a slightly different approach. It also has flow nodes just like GRs 1 and 2, but instead of interface nodes, which are modeling the

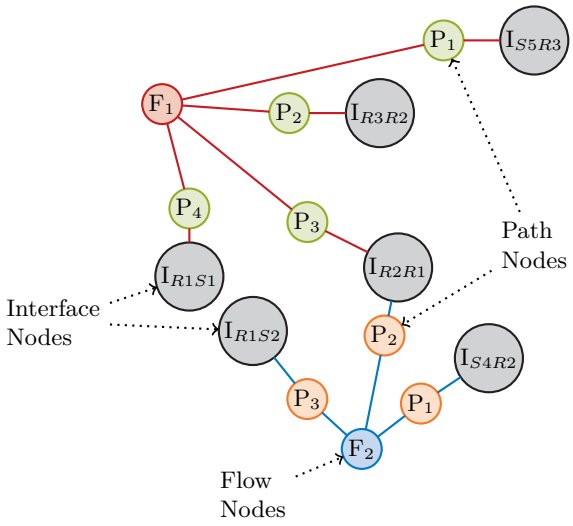


Figure 4.1: Graph Representation 2 of example topology from Figure 2.5

sending interfaces and the associated links, the network nodes are simply added to the GR as is. No individual sending interfaces are modeled, but the nodes as a whole. This means, that flow nodes are connected to the same network node in the GR, even if they do not pass over the same interface. In order to still be able to model the link properties, link nodes are added between the network nodes with the link's attributes that are held by the interface nodes in GRs 1 and 2. Because the interface nodes are not modeled separately, the queue utilization cannot be predicted using this GR, as multiple queues would have to share a single node. Figure 4.2 shows a visual representation of the transformed example topology from Figure 2.5.

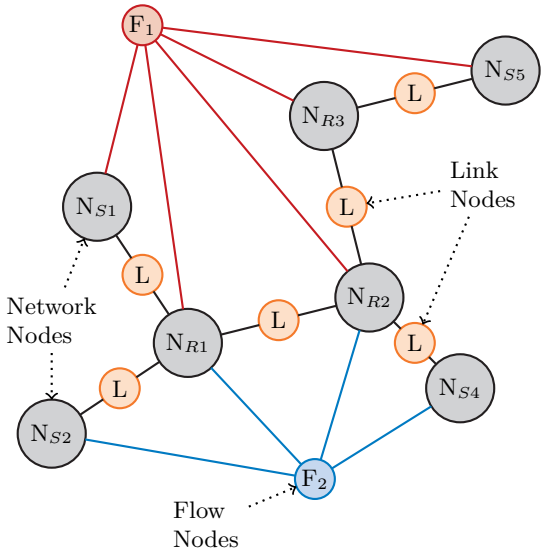


FIGURE 4.2: Graph Representation 3 of example topology from Figure 2.5

Chapter 5

Implementation

The workflow for training the neural network consists of four basic steps: The topology generation, the network simulation, the transformation of simulation results into GRs, and the actual training. This chapter gives an in-depth look at how these components were implemented.

5.1 Topology generation

The first step in the training pipeline is the generation of toplogies. Topologies are generated based on input parameters and output to a JSON file with a fixed structure. For an example file of a small network topology, see Listing A.1.

The input parameters are listed and described in Table 5.1. Internally, we differentiate between routers and servers. Flows are always established between two servers, and the routers are the interconnecting nodes between servers. A server is always connected to a single router, while routers can be connected to many routers and servers. Note that this differentiation is only done during the topology generation step. The output file (see Listing A.1) does not differentiate between servers or routers. Instead, they are listed as nodes and share the same numerical identifier space.

In order to simplify the simulation process, we decided to only generate tree-like networks for the scope of this thesis. The reason for this decision was, that in a tree, there is only a single route between two nodes. This allowed us to make use of the simulator's automatic static routing component without having to assign IP addresses at the generator stage. The route, which each flow traverses, is required in order to generate the GRs. If the flow could traverse multiple routes, the routing tables in the simulation would have

Parameter	Default Value	Description
`routers_min`	5	Minimum amount of routers in the topology
`routers_max`	12	Maximum amount of routers in the topology
`flows_min`	1	Minimum amount of flows in the network
`flows_max`	10	Maximum amount of flow in the network
`link_bandwidth_min`	10	Minimum link bandwidth in Mbit/s
`link_bandwidth_max`	100	Maximum link bandwidth in Mbit/s
`link_bandwidth_fixed`	*None*	Bandwidth in Mbit/s for all links, if set
`link_bandwidth_step`	10	Steps between bandwidth choices in Mbit/s
`link_delay_min`	1	Minimum link delay in ms
`link_delay_max`	50	Minimum link delay in ms
`link_delay_fixed`	*None*	Link delay in ms for all nodes, if set
`link_delay_step`	1	Steps between link delays in ms
`queue_size_min`	0	Minimum queue size in kB
`queue_size_max`	125	Minimum queue size in kB
`queue_size_fixed`	*None*	Queue size in kB for all nodes, if set
`queue_size_step`	1	Steps between queue sizes in kB
`sim_runtime`	60	Simulation duration in seconds

Table 5.1: Input parameters for topology generation script

to be manually filled to match the route proposed by the topology generation script. The framework could later be extended to include routing information in the generated topology, allowing arbitrary topologies to be simulated.

The topology generation algorithm works as follows: First, a random amount of routers between `routers_min` and `routers_max` are added to the topology. Note that all random numbers are chosen from a uniform probability distribution. Each router is assigned a node id in ascending order, and a `queue_size` in kB (either a random integer between `queue_size_min` and `queue_size_max` or `fixed_queue_size`). Then, links are added to form a random tree. For this, a random sequence of router node ids of length `len(routers) - 2` is generated. This sequence can be interpreted as a Prüfer sequence, which uniquely describes a tree. Using the Prüfer sequence, the links between the router nodes are added. Each link is assigned a random bandwidth between `link_bandwidth_min` and `link_bandwidth_max` or a fixed bandwidth of `links_fixed_bandwidth`. Note that the bandwidth is chosen in fixed steps. For example, with the defaults from Table 5.1, the generator would choose from the following set of possible bandwidths: $\{10\,\text{Mbit/s}, 20\,\text{Mbit/s}, 30\,\text{Mbit/s}, \ldots, 100\,\text{Mbit/s}\}$. In the

next step, a random number of flows between `flows_min` and `flows_max` is chosen. For each flow, two server nodes are added to the topology. The routers, to which they are connected, are chosen randomly while ensuring that they are not connected to the same router. The TCP implementation and the queue size of the server nodes are also chosen at random, and the links' attributes are as well.

Finally, the route for each flow is calculated using Djikstra's shortest path algorithm and stored in the flow objects. While calculating the routes, the traversed routers are marked as visited. Afterwards, all routers and links which are not traversed by at least a single flow are eliminated from the topology. Finally, the node ids are cleaned in order to accomodate for any eliminated routers and links. Then, the topology information is serialized to a JSON file and written to disk.

The output file contains three major lists with which the topology can be constructed: the list of nodes in the network, the list of links, and the list of flows between the nodes. It also contains the simulation duration in seconds, and a representation of the network in the *GraphViz* [17] format for debugging and visualization purposes.

5.2 Topology simulation

For the simulation component, we decided to use *ns-3* [2], a discrete event network simulator, because it ships with implementations for the most common TCP congestion control algorithms and has extensive documentation available.

The simulation component works as follows. First, the topology JSON file is parsed and the network nodes are instantiated. In ns-3, every component of a network is a subclass of the ns-3 *Node* type. Since ns-3 does not differentiate between routers and servers, the topology description file from the generator does not to either. These Nodes can have multiple ns-3 *Devices* attached to them. Devices in ns-3 can be thought of as network interfaces and can be of different types. For example, ns-3 provides devices for Carrier-Sense Multiple Access (CSMA) links, wireless links and Point-to-Point links using the Point-to-Point Protocol (PPP). It was decided to use the latter for the simulation, as the links from the topology generator only ever connect two nodes.

After creating the nodes, the nodes are connected using the links from the topology description, with their bandwith, delay and queue sizes set accordingly. After setting up the links, ns-3's `Ipv4GlobalRoutingHelper` is invoked to assign static IPv4 addresses to all interfaces and populate the nodes' routing tables.

Then, the flows are created. This is done by installing *Applications* on the ns-3 nodes. Luckily, ns-3 already has prebuilt application that fit the required use case, namely the

`TcpBulkSendApplication` and the `PacketSink`. The former sends as many bytes as possible to a specified destination address while the latter simply accepts incoming segments and discards them. Since the `PacketSink` is setup to use a TCP socket for listening, the `PacketSink` sends TCP `ACK` messages back to the `TcpBulkSendApplication`. The bulk-send application can also be customized to send a specified amount of data or to send only for a specified duration.

Finally, in order to obtain results from the simulation, ns-3's tracing functionality is utilized. Tracing makes it possible to subscribe to certain value changes, for example the RTT estimation of the TCP implementation, by supplying a callback function. The simulation program can operate in a verbose and a non-verbose mode. In the verbose mode, all collected metrics are written to a CSV file with a timestamp from the simulation. This is especially useful for debugging and plotting the values. The non-verbose output only aggregates the metrics for use in the simulation output file in order to save disk space. Since the simulations are deterministic, a simulation can be re-run at any time. A dedicated plotting and analysis script was also created in parallel to the simulation program, allowing the user to plot the data obtained from verbose simulations. There is also a script which plots information about an entire dataset of multiple simulations to check how different metrics in the dataset are distributed.

After the simulation has commenced, the traces of interest are aggregated. For example, for each flow, the average flow rate and RTT are computed. These aggregated values are then written to a new output JSON file, which contains the original topology input JSON in addition to these results.

It was found that simulation durations of 60 s provide a good balance between actual simulation runtime and the usefulness of the generated data for training. The simulation runtime scales linearly with the amount of events the simulator needs to process. Thus, longer simulation durations and higher throughputs in the simulated network increase the simulation runtime. It was decided to select a default value of 10 flows for the topology generation `flows_max` parameter and 100 Mbit/s as the default parameter for `link_bandwidth_max`, as this leads to reasonable simulation runtimes of around 10–20 min per topology on a testbed system. This allows us to generate enough data for training in a day on a system with 64 threads.

All steps leading up to this point were usually run on a testbed node with high CPU compute capabilities. Both the training data generation and simulation were parallelized using *GNU parallel* [18], which automatically assigns jobs to idle CPU cores.

Though ns-3 includes many different TCP implementations, we decided to limit our topologies to only include TCP Cubic and TCP Reno as valid implementations. Origi-

nally, it was planned to include TCP Vegas and TCP BBR as well. After inspection of the simulation results, it was decided to discard these simulations, as both TCP Vegas and TCP BBR behaved in an unpredictable manner in the simulator when in the same network with multiple flows. The simulator sometimes even reported the flows to have a higher bandwidth or smaller RTT than physically possible in the given network.

5.3 Graph Representation mapping

As discussed in Section 2.4, the data needs to be converted into a graph representation before training. A framework for training GGNNs using PyTorch and PyTorch-Geometric, an extension of PyTorch for graph-based machine learning, already existed, which was reused and adapted the existing code for this thesis. The existing framework provides a `Parser` class, which can be subclassed. Different subclasses can then generate different GRs.

To create a new GR, the `Parser` class is subclassed and its `process_data` function is overriden. This function has a `data` object parameter, which contains the results from the simulation, decoded to a Python dictionary. It is expected to return a *NetworkX* graph. NetworkX is a Python library for working with graph structures and allows embedding information in nodes and edges of the graph. These node and edge attributes are used to embed the node features of the network, including the ones to be predicted. The available attributes are supplied to the parser on instantiation. This attribute definition dictionary contains how the attribute is encoded (one-hot encoding vs scalar), a flag indicating whether the attribute shall be predicted, a masking list which is later used in the loss calculation to ignore non predicted attributes, and a normalization function with which all values of that attribute are normalized. At the time of writing, the existing framework does not yet support predicting different metrics at the same time. For example, it is currently not possible to train a single model which predicts both the TCP flow rate and TCP flow RTT at the same time. Instead, two models would have to be trained to predict both metrics. This is due to a limitation in the mask processing for the loss function, which could be resolved as future work. In practice, it is only a minor issue though, as training the model and inference does not take a long time for the topologies explored later on in Chapter 6 and the same simulation data can be used.

As discussed in Section 2.4, a decision was made to use minmax normalization. For this, a small script iterates over the whole dataset and gathers the minimum and maximum values for each features and writes them to an additional JSON file. This file can

be passed to the parser script as well, which can then use it for the normalizing the attributes.

After obtaining the graph representation as a NetworkX graph, the `graph2matrix` function can be called on the parser. This function is implemented in the `Parser` superclass and is the same for all GRs. It takes the NetworkX graph and transforms it into a PyTorch tensor akin to the matrix representation described in Section 2.4 and Figure 2.6b. The input and output features are stored in separate tensors, and a mask tensor containing the masking information is created as well. These tensors and metadata, such as the original NetworkX nodelist and node count, are encapsulated in a PyTorch `Data` object, which can be exported to a file using the NumPy `npz` file encoding format. The exporting process is completely handled by PyTorch.

5.4 Training

Because the framework was designed to construct GGNNs from `npz` files output by the `Parser` class from the previous section, minimal adjustments were required to make training work with the GRs used for this thesis.

Since PyTorch can utilize GPU compute resources to accelerate the training process, dedicated GPU testbed nodes were used for most trainings. The models we trained were fairly small when compared to the installed GPUs VRAM (3 GB compared to around 12 GB for most testbed GPUs), so it was decided to utilize a tool called *Neural Network Intelligence* (NNI) [19], which offers parameterized hyperparameter tuning and parallelized training on the same and even multiple GPUs (if present).

The script which instantiates and then trains the neural network is parameterized as well. The input parameters and their description can be found in Table 5.2.

The training script works as follows: First, the dataset is loaded. This is done using the PyTorch `DataLoader` class, which loads the `npz` files into memory and converts them back to PyTorch `Data` objects. The model itself is also instantiated, with the model class being chosen from mutiple different implementations. At the time of writing, two model architectures have been implemented: The `GGNN` model class implements the GRU approach from [4] and the `ResGGNN` class implements a GGNN using residual GRUs.

Then the dataset is split into a training and a validation dataset, with the ratio taken from the `train-test-split` parameter (see Table 5.2). The model is trained for the specified amount of epochs. After each epoch, the model is evaluated using the validation dataset from the split, and a MPE is calculated. This is done to obtain a metric which is comparable between different training runs even when different loss functions are used.

Parameter	Default Value	Description
`seed`	1	Seed used by RNG
`dataset`	*None*	Folder containing dataset `npz` files
`epochs`	15	Number of epochs to train
`learning-rate`	5×10^{-4}	Learning Rate (LR) of Adam optimizer
`weight-decay`	0	Weight decay rate of Adam optimizer
`lr-scheduler-factor`	5×10^{-4}	LR scheduler factor for Adam optimizer
`dropout`	0.5	Dropout used between linear layers
`dropout-gru`	0	Dropout used between GRUs
`train-test-split`	0.75	Fraction of dataset used for training
`batch-size`	16	dataset size of each batch
`hidden-size`	64	Size of the hidden states in the network
`nunroll`	10	Number of loops to unroll in the GGNN
`num-features`	*None*	Number of feature columns in the training data's x
`num-classes`	*None*	Number of feature columns in the training data's y
`regression`	*Not set*	Flag indicating whether a reggression shall be trained
`device`	`cuda`	Choice between `cuda` or `cpu`
`gradient-clipping`	`inf`	Value at which to clip the gradient
`model-architecture`	*None*	Choice between architectures, e.g. GRU or LSTM
`linear-layer-input`	*None*	Adds a linear layer before the memory cell if set
`num-layers`	1	Number of memory cell layers
`nni`	*None*	Flag that activates NNI framework integration
`loss-function`	`MSELoss`	Choice of Loss function from PyTorch
`last-layer-sigmoid`	*None*	Adds a sigmoid normalization after the last layer if set
`minmax`	*None*	Supply a minmax JavaScript Object Notation (JSON) file for denormalizat

Table 5.2: Input parameters for neural network training script

When the training is finished, the model is written to an output directory. When NNI is used, the model is not saved, as that would take up very large amounts of disk space very quickly. Instead, the model can be retrained by setting all the hyperparameters obtained from the NNI tuner using the training script input parameters.

5.5 Machine Learning model architecture

The framework developed at the chair was built with modularity in mind and allows the use of different architectures of GGNNs. At the time of writing, two different model architectures have been integrated into the framework: the `GGNN` architecture and the `ResGGNN` architecture. Both architectures are implemented in the form of two different Python classes, each being a subclass of the `MemoryCell` class, which in turn is a subclass of the PyTorch `Module` class. The `Module` is provided by PyTorch and provides the basic building blocks for constructing a Neural Network (NN). Subclasses of `Module`, including the ones presented below, override the `forward()` function, which, as the name suggests, is called on every foward pass of the NN.

5.5.1 GatedGraphConv

The `GGNN` class implements a model architecture closely resembling the ML model presented in Section 2.3. It uses the `GatedGraphConv` class provided by PyTorch-Geometric, which implements the PyTorch-Geometric `MessagePassing` interface. The `MessagePassing` interface is meant to model the message passing behaviour of GNNs where data from neighbouring nodes, and optionally edges, is aggregated. The `GatedGraphConv` class implements the GGNN cells with GRU cells for cell memory described in Section 2.3 and [10]. It also adds a few layers before and after the `GatedGraphConv` layers, depending on the NN configuration. If `linear_layer_input` is activated (see Table 5.2), a linear layer (`torch.nn.Linear`), a leaky Rectified Linear Unit (ReLU) layer (`torch.nn.LeakyReLU`), and a dropout layer (`torch.nn.Dropout`) with the parameter `dropout` from Table 5.2 are added to the network before the `GatedGraphConv` layers. As this is added in the `MemoryCell` class, these first layers are the same for the `ResGGNN` class.

After that follows a combination of a `GatedGraphConv` layer and a sequence of a LayerNorm (`torch.nn.LayerNorm`), leaky ReLU and dropout layers repeated by `num_layers` (see Table 5.2). The `GatedGraphConv` layer is initialized with the `nunroll` parameter, which indicates how many iterations of the message passing to unroll (See Section 2.3.3 for reference).

After the `GatedGraphConv` cell, a linear layer, a leaky ReLU layer, a dropout layer with `dropout`, and another linear layer are added. Finally, the output vector is passed through the logistical simgoid function if `last_layer_sigmoid` is set.

5.5.2 ResGatedGraphConv

The `ResGGNN` class has the same conditional linear input layers and output layers as the `GGNN` model class, as they both inherit those from `MemoryCell`. But in contrast to `GGNN`, `ResGGNN` uses the `ResGatedGraphConv` module provided by PyTorch-Geometric, which implements a residual GGNN as proposed in [20]. This architecture was not used for this thesis.

5.6 Validation and evaluation

In order to properly judge an ML model's accuracy, it is usually of interest to run it with samples it has not been trained or validated with before. These samples can be generated using the same steps as the samples used for training by simply adjusting the parameters of the topology generation. For the analysis of the results, scripts were created which take a folder or a single graph representation and a trained model as an input, predict the metrics using said model and then compute and plot the difference between the expected and the actual results. Possible metrics and plots of interest, which the framework can generate, include scatter plots of the expected and the predicted datapoints. These plots can also group datapoints based on other metrics, such as the TCP algorithm in use or the topology size in order to highlight possible causes of prediction errors. Other metrics, such as the relative error of predictions, is also calculated and plotted.

Chapter 6

Evaluation

In this chapter, different variations of the ML model are compared and evaluated for accuracy. Answers to the research questions stated in Section 1.2 are proposed.

6.1 Simulation and training setup

The following evaluation was done using the scripts and programs introduced in Chapter 5. For training, we generated multiple datasets with different parameters for topology size, flow count, queue sizes, etc. We utilized a script which generates, and afterwards, simulates a given amount of network topologies on a testbed node in parallel, in order to take advantage of high core counts. Since *GNU parallel* [18] was utilized, the amount of parallel simulations was automatically adjusted to the available compute resources. The simulations were run on a server with 64 threads.

6.1.1 Generated topologies

In total, 6 simulation batches were run in order to generate datasets for training and evaluation purposes. The parameters used to generate each dataset are listed in Table 6.1. For detailed documentation of default values and what each parameter is used for, please refer to Table 5.1.

Datasets D1 and D2 were used as a baseline for training and evaluation. Both were generated using the default parameters of the generator script. The resulting topologies are fairly small, with a maximum of 22 nodes per topology. The number of nodes in the topology depends on the randomly chosen number of routers and flows in the topology, as servers are added based on those two factors (refer to Section 5.1 for details). D3 was generated with the same parameters as D1 and D2, except for parameters concerning

Parameter	D1 & D2	D3	D4	D5	D6	Unit
Dataset Size	5000	5000	5000	5000	5000	–
`routers_min`	5	15	5	5	5	–
`routers_max`	12	25	12	12	12	–
`flows_min`	1	5	1	1	1	–
`flows_max`	10	15	10	10	10	–
`link_bandwidth_min`	10	10	10	10	10	Mbit/s
`link_bandwidth_max`	100	100	100	100	100	Mbit/s
`link_bandwidth_step`	10	10	10	10	10	Mbit/s
`link_delay_min`	1	1	1	1	1	ms
`link_delay_min`	50	50	50	50	50	ms
`link_delay_min`	1	1	1	1	1	ms
`queue_size_min`	–	–	50	–	10	kB
`queue_size_max`	–	–	200	–	200	kB
`queue_size_step`	–	–	1	–	1	kB
`queue_size_fixed`	125	125	–	25	–	kB
`sim_runtime`	60	60	60	60	60	s

Table 6.1: Generated datasets and the parameters used to generate them

the router and flow generation, in order to generate a dataset with larger topology sizes. The parameters of D4 and D6 were chosen to include a range of queue sizes and D5 was modeled to have a fixed but smaller queue size compared to D1 and D2.

Generating and simulating a 5000 topology dataset like D1 and D2 on aforementioned 32 core CPU nodes took around 5 h, while a dataset with larger topologies like D3 took 9 h.

In Figure 6.1, the distributions of the number of nodes (servers and routers) per topology are visualized for each dataset. As expected, all datasets except for D3 more or less share the same distribution with a minimum of 4 and a maximum of 22 nodes per topology. D3 has a minimum of 4 and a maximum of 47 nodes per topology.

Figure 6.2 shows a cumulative histogram of the path lengths of flows in the topologies of each dataset. Again, all datasets except for D3 share a similiar distribution, with an average path length of around 4.7 traversed nodes. Due to the larger topolgies in D3, the average path length is around 6.6 nodes here, with a maximum path length of 20 nodes.

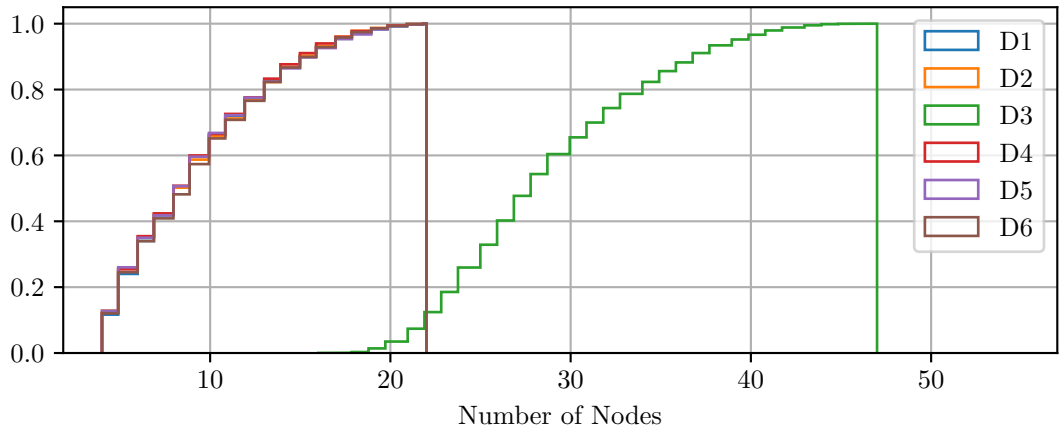


Figure 6.1: Cum. histogram of the number of nodes in the topo. by dataset

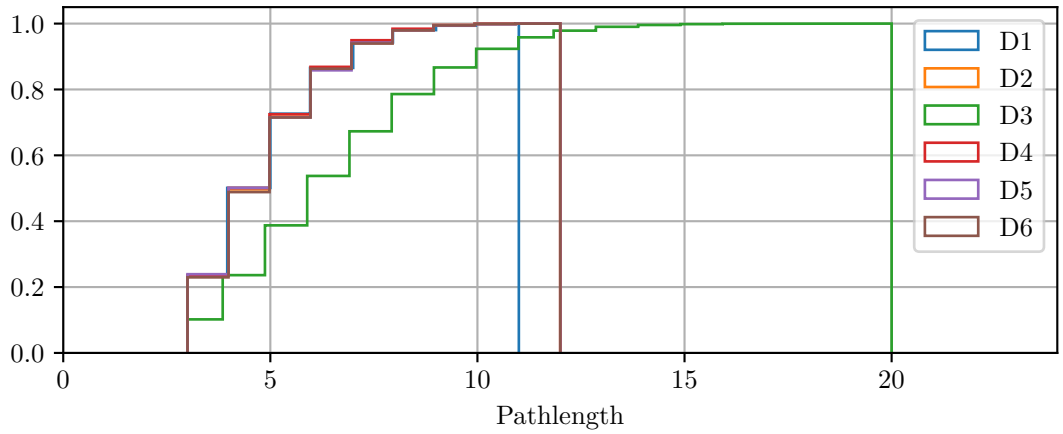


Figure 6.2: Cum. histogram of path lengths of flows by dataset

6.1.2 Distribution of simulation results

The datasets were all simulated on a testbed using ns-3 version 3.35 and the simulation program described in Section 5.2.

The resulting metrics from the simulation are plotted by dataset in Figures 6.3, 6.4 and 6.5. Figure 6.3 shows a cumulative histogram of the mean flow RTTs over each simulation runtime. As expected, the mean RTTs are larger in the dataset with larger topologies, and thus, longer path lengths. D1, D2, D4 and D6 share are very similiar distribution of mean RTTs, all capping around the 750 ms mark. The distribution of mean RTTs in D5 is slightly shifted to the left compared to D1, D2, D4 and D6, meaning the RTTs are a bit shorter. This makes sense, as the shorter queue buffers in D4 lead to

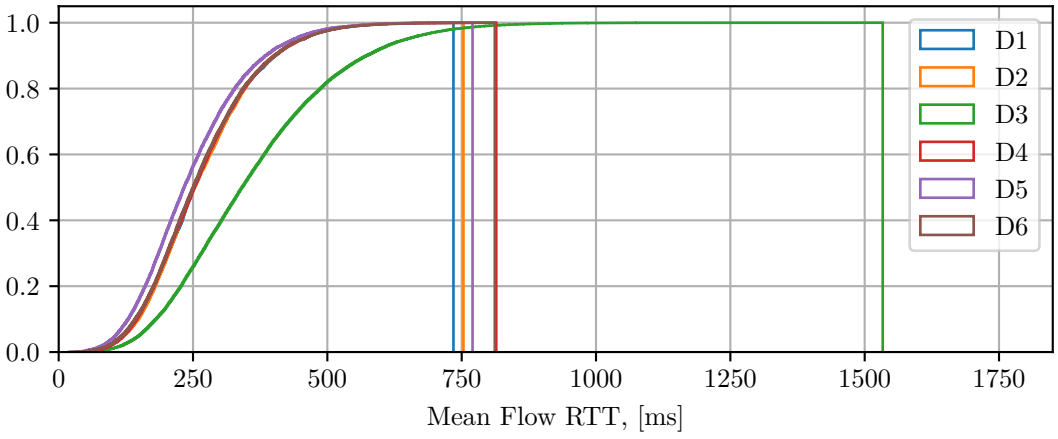


FIGURE 6.3: Cum. histograms of mean simulated flow RTTs of flows by dataset

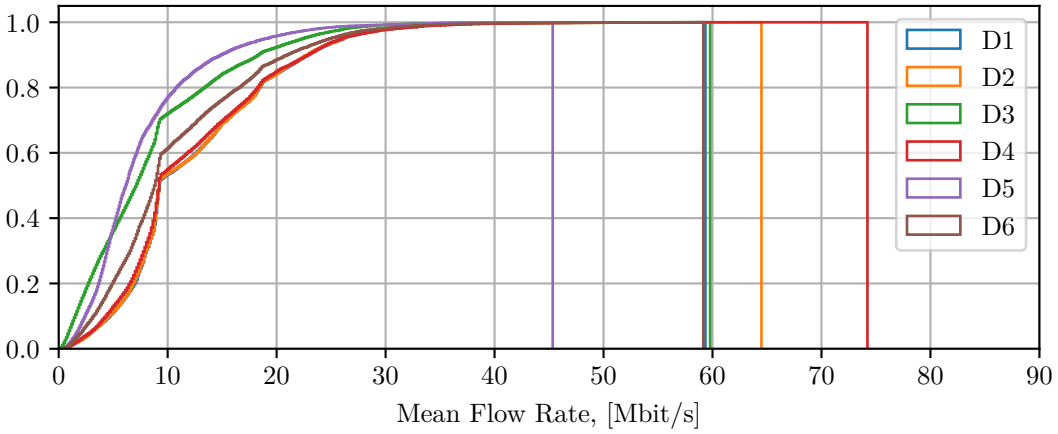


FIGURE 6.4: Cum. histograms of mean simulated flow bandwidths by dataset

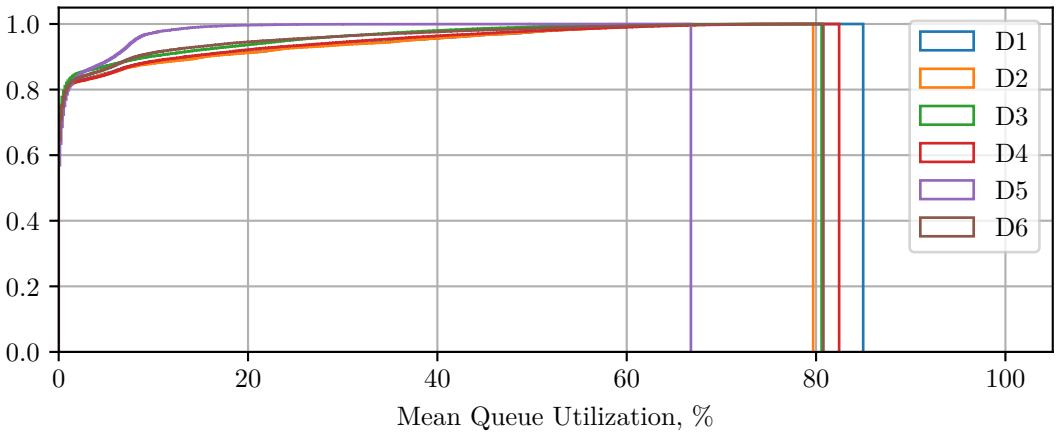


FIGURE 6.5: Cum. histograms of mean queue utilization per interface by dataset

reduced queueing delays. Looking at Figure 6.4, the mean TCP flow rates in D3 and D5 are also significantly smaller than in the D1, D2 and D4 due to the higher path lengths and shorter queue sizes, respectively.

The maximum mean flow rate of D5 (45.3 Mbit/s) is also reduced compared to the maximum flow rates of the other datasets, even D3, ranging from 59.7 Mbit/s to 74.2 Mbit/s. There is also a noticable spike at the 10 Mbit/s mark and a smaller spike at the 20 Mbit/s mark in all histograms except for D5. These are due to `link_bandwidth_min` and `link_bandwidth_step` being set to 10 Mbit/s each, and probably produced by a large number of flows where this smallest possible link bandwidth constitutes the bottleneck link of the flows.

Figure 6.5 shows cumulative histograms of the mean queue utilization expressed in percent separated by dataset. The mean queue utilizations are measured at each sending interface in the simulation over the course of the entire simulation. Overall, the mean queue utilizations are somewhat low across all datasets, with the average mean queue utilization lying around 3 % to 4 % for D1, D2, D3, D4 and D6. Only D5 has an even lower average mean queue utilization of 1.3 % and a maximum mean queue utilization of only 67 % compared to maxima of around 81 % for all other datasets. The reason for this is, that every flow only ever has a single bottleneck link. The other queues on the flows' paths stay mostly empty, leading to many interfaces with low queue utilization.

6.2 Trained models and results

In order to evaluate the GGNN ML model for answering the research questions presented in Section 1.2, parsers for the GGNN framework for the different GRs presented in Chapter 4 were implemented. Since the framework does not allow for predicting multiple metrics simultaniously, three parsers were created for each GR, except for GR 3. GR 3 is not suitable for learning queue utilization as it embeds no dedicated interface nodes to track queue utilization with. Thus, it was omitted for the queue utilization metric.

Then, a model was trained with data generated using D1 and each parser, creating a total of eight models created using D1 (3 GRs $\times$ 2 flow metrics + 2 GRs $\times$ 1 interface metric). The training for each model was done using NNI (see Chapter 5 for details) with 300 trials for hyperparameter tuning per model.

The training was done on a testbed node with an 8 core / 16 thread Intel Xeon E5-2620 v4 CPU and 4 Nvidia GTX 1080 Ti GPUs. The training was parallelized with up to 24 trials running in parallel across all 4 GPUs. Running more trainings simultaniously was not possible due to memory constraints both on the CPU and GPU side. Each

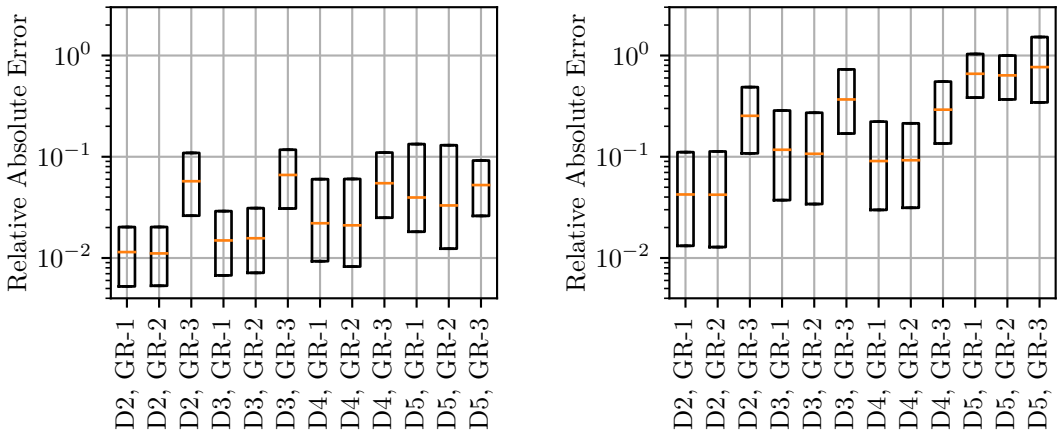
NNI trial took between 7 min and 15 min, depending on the model. For each model, the best performing of the 300 trial models was selected and retrained using the obtained hyperparameters in order to obtain the PyTorch model for inference use. An inference and plotting script was used which performs inference for a given dataset using the given model and outputs the results to both CSV files and plots (see Section 5.6 for details). The inference script manages to process the data of a single dataset (5000 topologies) in 12–20 s, depending on the model and topology size in use. These metrics were obtained on the same system the models were trained on.

In Figures 6.7, 6.8 and 6.9, the results of mean flow RTT, mean flow rate and mean queue utilization predictions by the models created from D1 for datasets D2 through D5 are visualized as scatter plots, where each point represents a single metric predicted using the respective model. The horizontal axis of the scatter plots corresponds to the value reported by the model, while the vertical axis corresponds to the expected value obtained by simulation. The dashed diagonal gray line represents the theoretical optimum, where the prediction exactly equals the expected value. The points are also colorized differently in the scatter plots of flow rate and flow RTT depending on the TCP congestion control algorithm employed by the flow corresponding to the datapoint in order to highlight possible differences in prediction accuracy. Each row of graphs corresponds to a single dataset, while each column of graphs corresponds to one GR. If a datapoint is above the diagonal, it means that the value predicted by the model was too low. Likewise, if a point is below the diagonal, it means the value predicted by the model was too high.

Figure 6.6 shows the absolute relative error for all combinations from Figures 6.7 and 6.8, while Table 6.2 shows the mean absolute relative error for all combinations. The absolute relative error is calculated in the following way:

$$\epsilon = \frac{|y - y'|}{y} \tag{6.1}$$

with ϵ being the absolute relative error, y being the expected value and y' being the predicted value.



(a) RTT

(b) Flow rate

Figure 6.6: Comparison of the absolute relative errors of RTT / flow rate prediction separated by dataset and GR. Boxes represent the first quartile and third quartile, orange lines represent the median / second quartile. The vertical axis is scaled logarithmically.

Since the mean queue utilization was sometimes zero and very often near zero in the datasets, a relative error does not make much sense for comparing the accuracy, as otherwise tiny differences can result in verly large relative error values, while an expected value of zero even causes division by zero issues. We decided to use the absolute error

$$\epsilon = |y - y'| \tag{6.2}$$

for comparing the queue utilization results instead.

Table 6.3 shows the mean absolute error for all combinations from Figure 6.9. Note that these are not to be confused with relative errors (like in Table 6.2), but that the unit of the mean queue utilization is percent. For example, the 1.4 % mean absolute error for D2 and GR 1 means, that on average, the error when predicting D2 with the GR 1 model is about ± 1.4 %.

6.2.1 Comparison of graph representations

The following paragraphs present and analyze the results from the performed model evaluation and explain the scatterplots from Figures 6.7, 6.8 and 6.9 in detail.

Round-Trip Time

D2 represents optimal conditions for models trained using D1, as they more or less share the same distributions of topology size, bottleneck bandwidths, etc. Looking at the three scatter plots for the RTT prediction of flows in D2 (Figures 6.7a, 6.7b, 6.7c)

	GR 1	GR 2	GR 3
D2	1.66 %	1.68 %	7.85 %
D3	2.30 %	2.61 %	8.60 %
D4	4.40 %	4.37 %	8.10 %
D5	10.08 %	9.73 %	7.18 %

(a) RTT

	GR 1	GR 2	GR 3
D2	9.69 %	9.57 %	45.49 %
D3	25.28 %	21.55 %	90.76 %
D4	16.69 %	16.19 %	47.85 %
D5	83.84 %	81.48 %	108.70 %

(b) Flow rate

Table 6.2: Mean absolute relative errors for predictions of mean flow RTT and flow rate using different datasets by models trained using D1 and GRs 1 - 3

	GR 1	GR 2
D2	1.40 %	0.80 %
D3	0.90 %	0.80 %
D4	1.80 %	1.50 %
D5	2.60 %	2.80 %

Table 6.3: Mean absolute errors for predictions of mean queue utilization in percent using different datasets by models trained using D1 and GRs 1 and 2

and the associated absolute relative error plots in Figure 6.6a, one can see that GRs 1 and 2 perform significantly better than GR 3. The mean absolute relative error lies at 1.66 % and 1.68 % for GRs 1 and 2 and at 7.85 % for GR 3. Here, GR 1 performs best, predicting metrics which are accurate to 5 % of the actual value 94.9 % of the time and accurate to 10 % of the actual value 98.9 % of the time. The scatter plots 6.7a and 6.7b show that the models with GR 1 and GR 2 perform better for lower RTTs, while no significant difference between TCP Cubic and TCP Reno can be found in this case. This makes sense, as both algorithms are loss based and should behave similiarly in this scenario. The model using GR 3 shows a significant offset above the optimum line (values are predicted too low) and an overall wider spread from the optimum line. This probably happens, because of the way the interfaces are rerpresented in GR 3. When multiple flows traverse the same network node on different interfaces, the model using GR 3 will not be able to differ and predict lower RTTs, although the flows do not interfere with each other. This leads to overall higher predicted RTTs.

In the next row of scatter plots (Figures 6.7d, 6.7e, 6.7f), the results for applying the model to D3 are plotted. Again, GR 1 performed best with a mean absolute relative error of 2.30 % compared to 2.61 % and 8.60 % for GRs 2 and 3, respectively. However, this error gets significantly larger in the range above 750 ms. When looking only at datapoints where the expected value is larger than 750 ms, the mean absolute relative

errors increase to 8.38 %, 17.39 % and 20.2 % for GRs 1, 2 and 3. The reason for this is, that the model was trained using the D1 dataset, which caps out at mean RTTs of around 750 ms, leading to predictions which are too low, as the model has never encountered RTTs this high. The plot for GR 2 shows a harder cutoff than the one for GR 1. This can be explained with the model's hyperparameters, as the NNI tuner chose to set `last_layer_sigmoid` for the GR 2 model, while leaving it unset for the GR 1 model. The sigmoid function caps its output to a value between 0 and 1, thus limiting the values the model can predict to the normalization maximum. As we will show later in Section 6.2.3, the graph looks a lot more like the one with GR 1 when adjusting the model's minmax bounds.

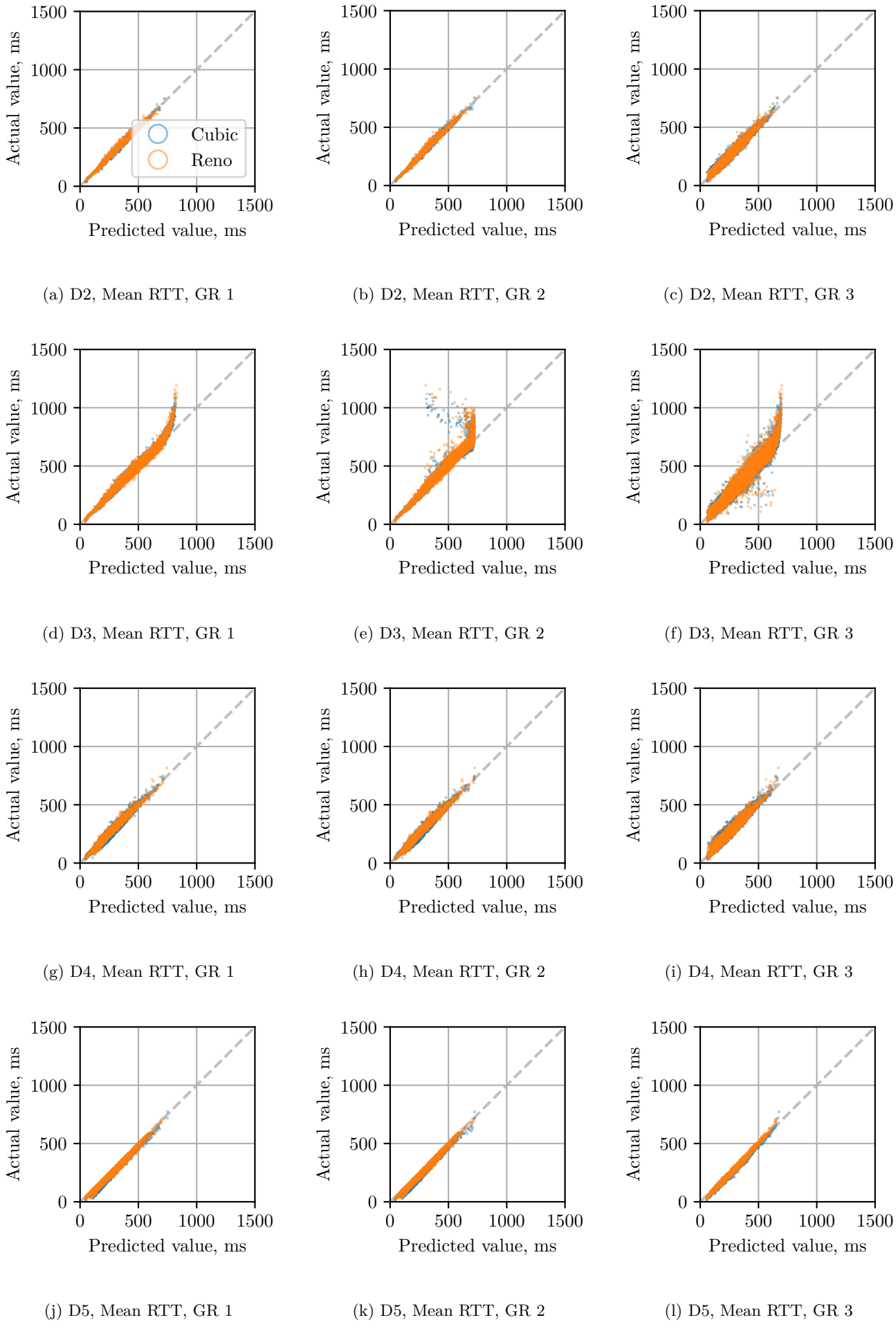
The models applied to D4 yield a mean absolute relative error of 4.40 %, 4.37 % and 8.10 % for GRs 1, 2 and 3, respectively. Overall, all three GRs perform slightly worse for D4 than for D2, as is to be expected due to non static queue sizes which the model was not trained with. When looking at the scatter plots (Figures 6.7g, 6.7h, 6.7i), a notably larger spread is visible. Also, the spread is slightly larger for TCP Cubic flows. Considering that the model was not trained with varying queue sizes, the model performs surprisingly well for the D4 dataset.

Finally, the results from applying the model to D5 result in a mean absolute relative error of 10.08 %, 9.73 % and 7.18 % for GR 1, 2 and 3, respectively. Curiously, this is the only dataset where the model trained with GR 3 performs better than both other compared GRs, even if only by a small amount. The scatter plots for the results from the models trained using GR 1 and 2 (Figures 6.7j and 6.7k) also show a distinct amount of datapoints offset below the diagonal (predicted value is too high), which is not present in the scatter plot for GR 3 (Figure 6.7l). These results are also somewhat expected, as the model has never encountered fixed very small queue sizes before, and thus, can't generalize well in this regard.

The difference in mean absolute percentage error between the two different TCP congestion control algorithms is less than 2 % for all dataset combinations, except for D5, where the difference reaches 4.8 %. Here, TCP Reno flows' RTTs are predicted more accurately. Otherwise, the differences between the two TCP algorithms are insignificant.

Flow rate

The TCP flow rate predictions of dataset D2 with the models obtained using D1 and the three GRs are vastly different in terms of accuracy from their RTT counterparts. Here, the mean absolute relative error for the best case scenario (D2, GR 1) is 9.57 % for GR 2. With this model (GR 1), the flow rates in D2 are predicted accurate to 5 % of the expected value 54.4 % of the time and accurate to 10 % of the expected value



(a) D2, Mean RTT, GR 1

(b) D2, Mean RTT, GR 2

(c) D2, Mean RTT, GR 3

(d) D3, Mean RTT, GR 1

(e) D3, Mean RTT, GR 2

(f) D3, Mean RTT, GR 3

(g) D4, Mean RTT, GR 1

(h) D4, Mean RTT, GR 2

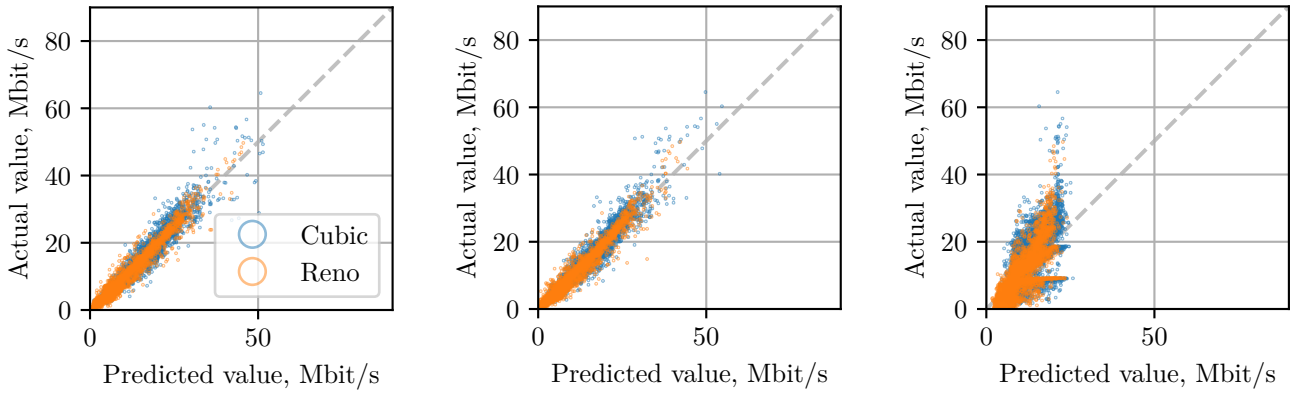
(i) D4, Mean RTT, GR 3

(j) D5, Mean RTT, GR 1

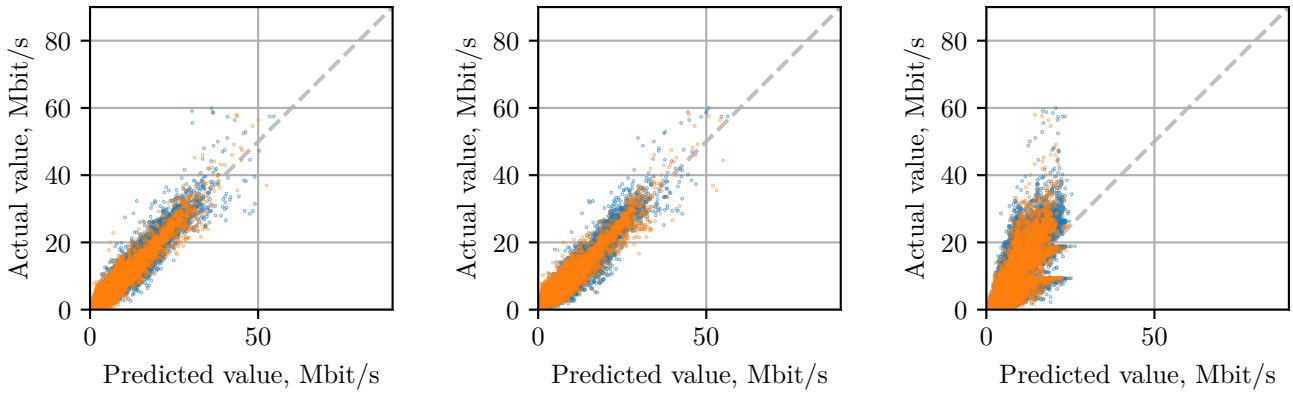
(k) D5, Mean RTT, GR 2

(l) D5, Mean RTT, GR 3

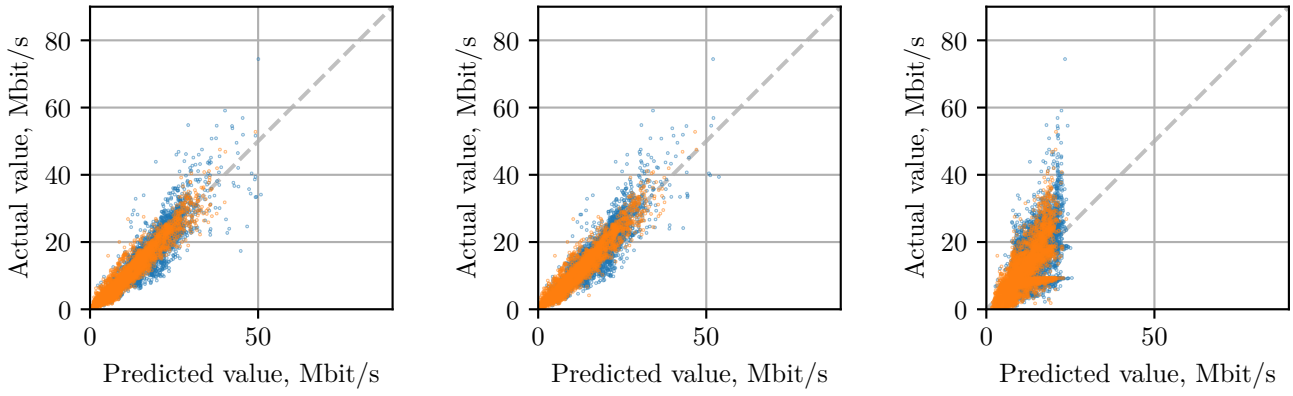
FIGURE 6.7: Scatter plots of RTT prediction of D2 - D5, all trained using D1



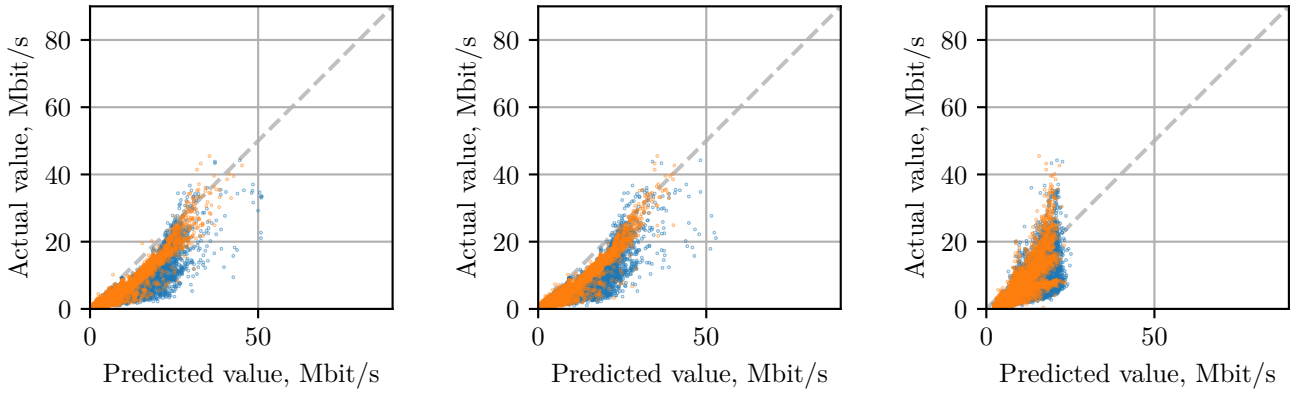
(a) D2, Mean flow rate, GR 1 (b) D2, Mean Flow Rate, GR 2 (c) D2, Mean Flow Rate, GR 3



(d) D3, Mean Flow Rate, GR 1 (e) D3, Mean Flow Rate, GR 2 (f) D3, Mean Flow Rate, GR 3



(g) D4, Mean Flow Rate, GR 1 (h) D4, Mean Flow Rate, GR 2 (i) D4, Mean Flow Rate, GR 3



(j) D5, Mean Flow Rate, GR 1 (k) D5, Mean Flow Rate, GR 2 (l) D5, Mean Flow Rate, GR 3

FIGURE 6.8: Scatter plots of flow rate prediction of D2 - D5, all trained using D1

72.3 % of the time. While the scatter plots for GR 1 and 2 (Figures 6.8a and 6.8b) look somewhat promising, the plot for GR 3 (Figure 6.8c) shows significant diversion from the optimum, with a maximum absolute error of 44.8 Mbit/s. The model with GR 3 also shows noticable plateaus at the 10 Mbit/s and 20 Mbit/s marks (expected values, y axis), where the model predicted higher flow rates. These are probably caused by the spikes in the distribution of flow rates from the simulation output discussed in Section 6.1.1. The GR 3 model fails to predict values greater than 30 Mbit/s, while GR 1 and 2 manage to do so, albeit with decreased accuracy (mean abs. rel. error for (D2, GR 1) $<$ 30 Mbit/s: 9.62 %, $\geq$ 30 Mbit/s: 12.52 %). This is also caused by the distributions of flow rates from the simulation data of D1, as 97 % of flow rates in D1 are smaller than 30 Mbit/s.

As expected, the results for D3, D4 and D5 are worse in terms of accuracy than the ones for D2, with D4 having the lowest mean absolute relative error of 16.19 % using the model with GR 1. All of the plots for the combination of GR 3 and these three datasets (Figures 6.8f, 6.8i and 6.8l) show the same plateaus and cutoff at the 25 Mbit/s predicted value mark as the one with D2, suggesting that is is a general issue with GR 3.

The mean flow rates from D4 are similiarly distributed to the ones from D2, albeit being a bit lower across the whole dataset (12.2 Mbit/s vs. 12.4 Mbit/s mean respectively), leading to a slighty higher but still comparable mean absolute relative error of around 16.4 % for the models using GR 1 and 2 (Figures 6.8g and 6.8h).

The models obtained using GRs 1, 2 and 3 combined with dataset D5 yield the worst results in this trial, with an mean absolute relative error of over 108 % in the worst case. As the scatter plots show, the models overestimate the mean flow rate drastically, with nearly all points in the scatter plot below the diagonal (see Figures 6.8j, 6.8j and 6.8j). Especially, TCP Cubic flows are estimated by the model to have significantly higher flow rates than they actually do according to the simulation.

Except for D5, the flow rate of TCP Cubic flows is predicted with greater accuracy than that of TCP Reno flows. For the best case scenario (D2, GR 2), the difference in the mean absolute percentage error is 3.7 % (Cubic is predicted with lower error) between the two flow types. The maximum difference occurs for the combination (D3, GR 3), where there is a difference of 71 % between the mean absolute relative errors between the two flow types in favor of TCP Cubic.

Queue utilization

As stated previously, the models for testing the performance regarding the mean queue utilizations at the sending interfaces was only done using GRs 1 and 2, as GR 3 does not

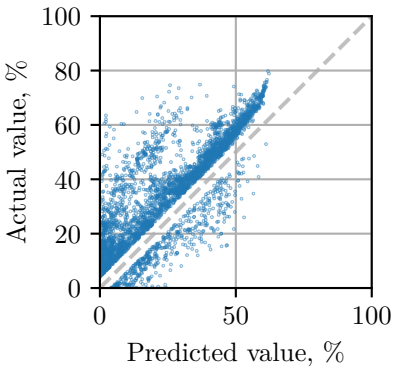
model individual interfaces. The results are displayed in Figure 6.9 as scatter plots and the mean absolute errors are given in Table 6.3. We adjusted the scatterplots to only include the top 10 % of errors in order be able to gauge at the origin of the outliers better. As Table 6.3 shows, the mean absolute errors are relatively small, as many datapoints have low queue utilization and are predicted with relatively high accuracy. Compared to the flow metrics, the queue utilization results show a much greater amount of outliers than the graphs for the other metrics from the previous paragraphs.

The results for datasets D2, D3 and D4 all exhibit a similiar pattern, with larger spread around the optimum line in GR 1 compared to GR 2. The first GR shows a noticably larger cloud of datapoints above the optimum line where the predicted value is too low. Both models struggle with values near zero in all datasets, as shown by a large amount of datapoints near the origin of both axes. A vertical line around the 20 % mark can also be found in the scatterplots for the GR 2 model for D2, D3 and D4. It is best visible in Figure 6.9f, but the cause is unclear, as it does not correspond to any specific value in the original simulation data. All plots show a slight skew above the diagonal optimum line for higher queue utilizations, where the datapoints are consistently predicted to be too high.

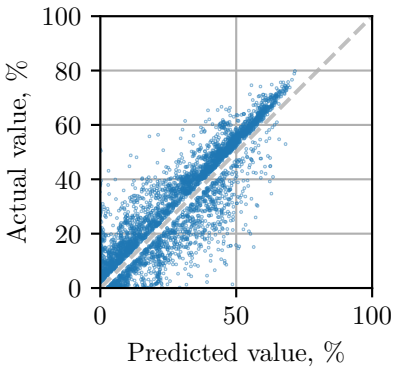
Looking at the first row of graphs, the models trained with D1 perform best with the D2 dataset because they share the same distributions of metrics. In the best case scenario (D2, GR 2, Figure 6.9b), a mean absolute error of 0.8 % is achieved. The plot for GR 1 has far more outliers above the diagonal compared to GR 2.

The plots for D3 look similiar to the plots for D2, but the amount of outliers further away from the optimum diagonal is noticably larger. The empty line at the optimum looks smaller and suggests lower errors in the 90th percentile range (which was ommitted). This is due to a higher amount of lower queue utilization datapoints in D3. When looking closely, the slope of the line where most datapoints reside seems to be steeper in Figure 6.9c compared to Figure 6.9d. At first, we believed that this was also because of the `last_layer_sigmoid` hyperparameter, but this is not the case. The `last_layer_sigmoid` hyperparameter flag was not set for both queue utilizattion models shown here. Apparently, the inclusion of path nodes in GR 2 simply leads to better accuracy for queue utilizations higher than the ones encountered during training.

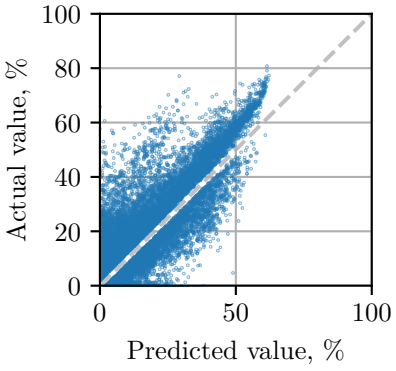
Figures 6.9e and 6.9f show the scatter plots for the results for dataset D4. The error corridor, where the 90th percentile of errors would reside, is larger compared to results from D2 and D3, indicating a bigger deviation from the optimum. As expected, the number of outliers increases again compared to D2 and D3, but more outliers are below



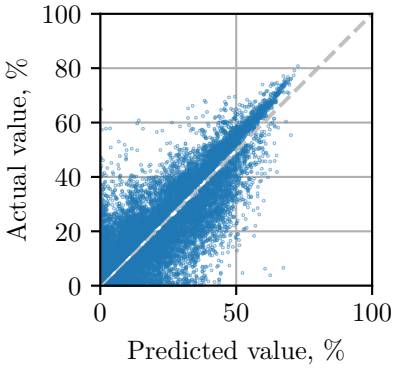
(a) D2, Mean Queue Util., GR 1



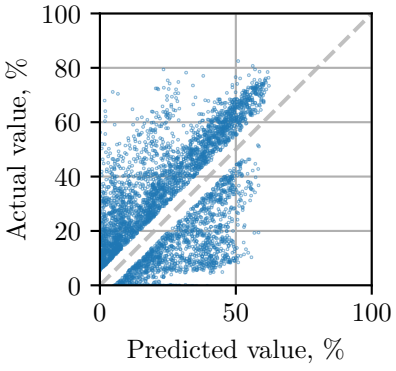
(b) D2, Mean Queue Util., GR 2



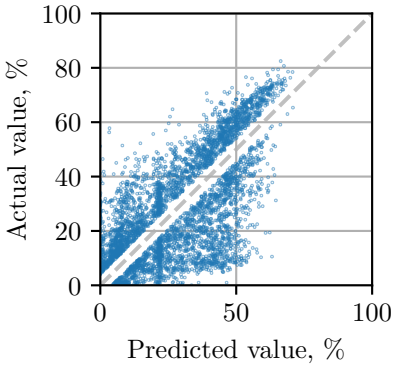
(c) D3, Mean Queue Util., GR 1



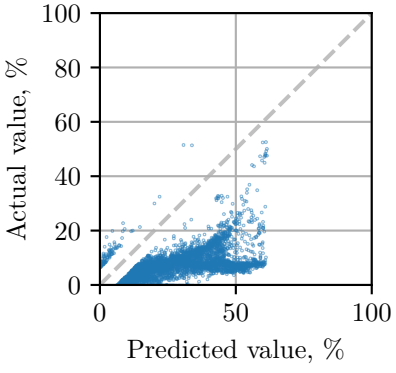
(d) D3, Mean Queue Util., GR 2



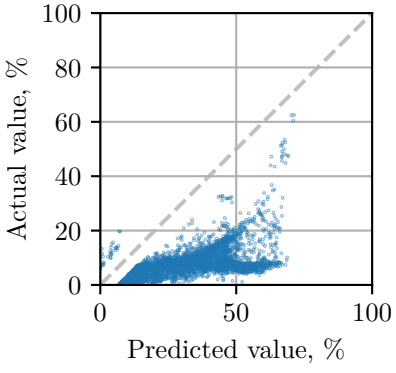
(e) D4, Mean Queue Util., GR 1



(f) D4, Mean Queue Util., GR 2



(g) D5, Mean Queue Util., GR 1



(h) D5, Mean Queue Util., GR 2

FIGURE 6.9: Scatter plots of queue util. prediction of D2 - D5, trained using D1, 90th percentile removed

48

the optimum diagonal. This is due to D4 having variable queue sizes, which the model from D1 has not been trained with.

Finally, the scatter plots for D5 (Figures 6.9g and 6.9h) show, that both models completely overestimate the queue utilization when encountering queue sizes which are drastically smaller than the one the model was trained with, resulting in larger absolute errors where the queue utilizations are consistently predicted too high.

When ignoring the datapoints where the expected value is zero, a relative error can be calculated. For the best case scenario (D2, GR 2), this results in the model having a mean relative error of 131.6 %, which is not very insightful, because the expected values for queue utilization in D2 are very small (median queue utilization is 0.016 % when removing all zero datapoints from D2).

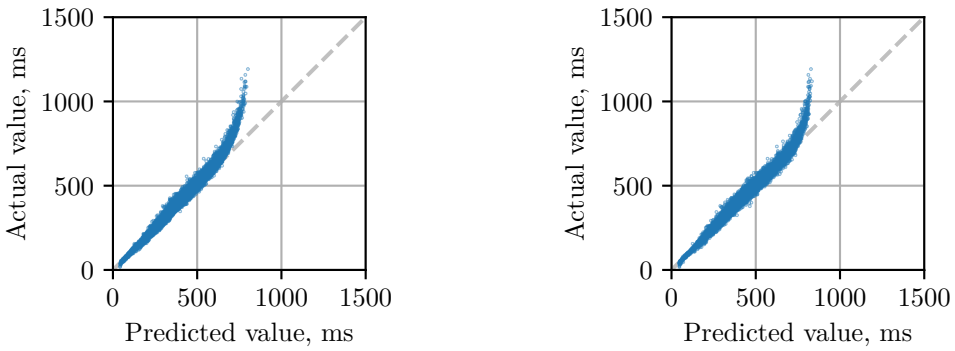
6.2.2 Generalization to larger network sizes

As was already shown in the previous subsection, the model is able to predict metrics from larger topologies than the ones it was trained with (see results of model trained with D1 predicting metrics of D3, Figures 6.7d, 6.8d), but looses accuracy once the expected values are too far off from the original range of expected values from the training dataset.

In order to check whether this happens due to normalization with the minmax range from D1, the model for predicting the flow RTT with D1 and GR 1 was retrained with a different minmax range, this time utilizing the minimum and maximum over both D1 and D3. Figure 6.10a shows the results of this experiment. The results indicate that this is simply a problem with unknown input data and has nothing to do with the normalization, as the mean absolute relative error increases to 2.96 %, while the scatter plot looks almost identical to the original in Figure 6.10b.

We also wanted to verify our claim about the `last_layer_sigmoid` function causing the hard cutoff in Figure 6.7e. We retrained the (D1, GR 2) model using the same hyperparameters, but supplied new minmax bounds for training. The result can be seen in Figure 6.11a next to the original graph for comparison. The graph looks more like the one with GR 1, with fewer outliers and a slight increase in the mean absolute relative error to 3 %.

The mean absolute error for all datapoints in D3, where the expected value is larger than the maximum RTT of D1, is 8.4 % for the GR 1 model and 14.1 % for the GR 1 model with the adjusted minmax normalization range. The mean absolute error of 2.82 % of the minmax adjusted model over the whole of D3 is also slightly worse compared to the 2.30 % of the non-adjusted model. The larger error margin does not correlate much with the overall topology size, but rather with the path length of the individual flow.



(a) minmax(D1, D3)

(b) minmax(D1)

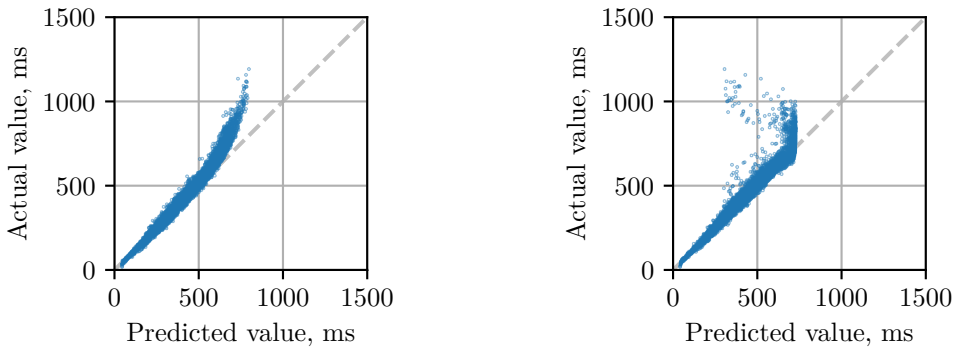
Figure 6.10: Scatter plots of prediction of D3 RTT using D1, GR 1, different minmax bounds

For a visualization of the (D3, GR 1) prediction error, see the two plots in Figure 6.12. On the left (Figure 6.12a), the absolute relative error ist shown in relation the the path length of the associated flow. Please note that there are less samples for the longer path lengths to the right of the graph, leading to smaller deviations in the errors. On the right side (Figure 6.12b), the scatterplot from Figure 6.7d is colored by path lengths. Using this information, it seems clear that the large deviation is mostly caused by path lengths which are greater than what is encountered in the training dataset D1.

While evaluating the results, we noticed that the graphs resulting from GRs 1 and 2 could be disconnected, for example when two flows pass over the same router, but different interfaces. We compared the mean absolute relative error between datapoints from graphs which are fully connected to those which are not. The difference is less than 1 % for all RTT model and dataset combinations, except for D5, which showed a difference of 1.5 % favoring the fully connected topologies. The difference is larger for the flow rate models, with a maximum difference of 5.8 % for (D3, GR 3), favoring the not fully connected topologies.

6.2.3 Generalization to different queue sizes

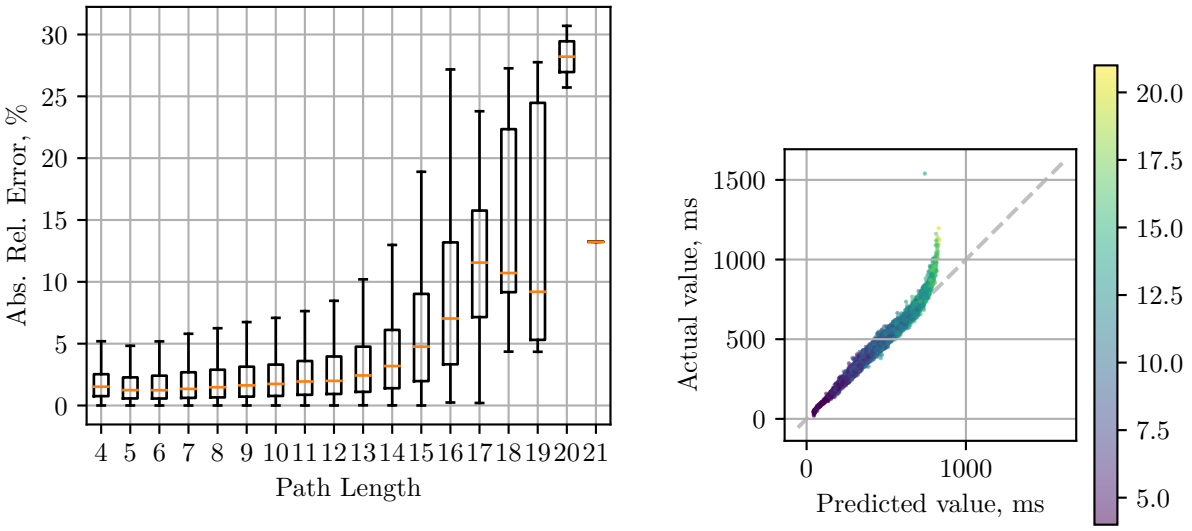
As discussed in the previous sections, the large error increase when adjusting the queue sizes at the interface nodes is due to the model trained with D1 expecting fixed queue sizes of 125 kB. In order to test how the model would perform if it had experienced differing queue sizes, we created a new training dataset, D6. D6 shares all properties of D4, except with a broader range of queue sizes. These were chosen randomly from the



(a) minmax(D1, D3)

(b) minmax(D1)

Figure 6.11: Scatter plots of prediction of D3 RTT using D1, GR 2, different minmax bounds



(a) Abs. rel. error by path length. Boxes represent the first quartile and third quartile, orange lines represent the median / second quartile.

(b) Colors indicate Path Length

Figure 6.12: Visualization of prediction error of D3 RTT using model D1, GR 1

range of 10 kB to 200 kB. The range was chosen so that D6 would have some topolgies similiar to the ones in D5.

We then trained a model with GR 1 and D6 for predicting the mean flow RTTs, again using 300 trials using NNI. This time, the training data (and validation data) was normalized using the minmax bounds of D6. After training, we performed inference

against D2, D3, D4 and D5 which were also normalized using the minmax bounds of D6 for this task.

The inference against D2 yielded a mean absolute relative error of 1.81 %, which is only slightly higher than the error of the model obtained with D1 and GR 1 for the RTT. This makes sense, as there are more parameters in the model but the queue sizes in D2 are constant. A model expecting a fixed queue size will perform better on a dataset with fixed queue sizes than a model expecting varying queue sizes.

When inferring the RTTs for D3, the model yields a mean absolute error of 2.67 %, which is also slightly higher than the one obtained using the D1 model (2.3 %). This was also expected for the same reason as with D2 presented above.

Applying the model for predicting D4 leads to better results than with the model obtained from D1. The mean absolute relative error is reduced to 2.25 %. That the model would perform better is to be expected, but the magnitude of this change is rather low. The model obtained from D1 managed to achieve a mean absolute relative error of 4.4 %, which is not that far off compared to this model.

Finally, comparing the model against D5 is where the inclusion of the low queue sizes in the topologies really makes a difference. The model predicts the mean RTT in D5 with a mean absolute error of 1.13 %, nearly an order of magnitude better than the model obtained from D1 (10.08 %). Figure 6.13 illustrates this difference in accuracy.

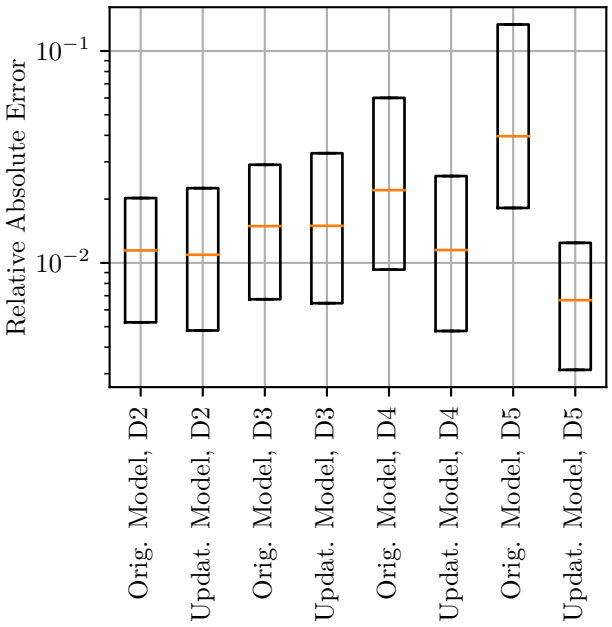


Figure 6.13: Comparison of absolute relative prediction error of RTTs using models obtained using D6 (Updated) and D1 (Original). Boxes indicate first and third quartile, orange lines mark the median/second quartile

6.2.4 Analysis of feature importance

When looking at the different GRs, it becomes clear that the features embedded in the GR from the original network topology play a great role in determining the model's accuracy.

The main difference between GRs 1 and 2 lies in the inclusion of path nodes, which add the information of direction of the flow to the model. By inspecting the results, it appears that including this metric has mostly small but negative effects on the accuracy of the flow RTT and flow rate predictions, and postive but relatively small effects on the prediction accuracy of queue utilizations.

By comparing the results from GR 3 with GRs 1 and 2, it is obvious that the modeling of individual queues and their respective sizes has a great effect on the model's performance when predicting flow RTTs and flow rates. For example, the error differs by a factor of 5 when comparing (D2, GR 1) and (D2, GR 3). The scatter plots associated with GR 3 show, that not modeling individual interfaces results in the metrics being predicted too low once multiple flows traverse the same network node on different interfaces.

6.3 Summary of evaluation results

The results from this evaluation show, that GGNNs are suitable for the general prediction of TCP performance metrics. However, it also becomes clear that a "one size fits all" approach in regards to different GRs and the predicted metrics could be disadvantageous. Different metrics seem to benefit from different GRs, as shown by the difference in RTT and flow rate predictions.

We showed that the models can learn how certain metrics of TCP flows behave in arbitrary networks, with multiple flows interacting with one another. As expected, the models' loose some accuracy when facing previously unknown data, such as larger topologies. However, the models did still perform relatively well with data where the expected results are atleast somewhere in range of the training data. For example, a mean absolute relative error of under 10 % for model (RTT, GR 1) when predicting D3 which has never encountered topologies this large is still a viable metric, depending on the application. The same goes for different queue sizes. As we show in Section 6.2.3, the models benefit from learning from a broader range of data which is somewhere near the range of the data which one wants to predict later on.

We found that the model's hyperparameters have a great impact on the models accuracy. For example, the `last_layer_sigmoid` hyperparameter in combination with minmax normalization can cause the model to be unable to predict values outside of the min-max bounds it was trained with. If generalization to larger topologies is a goal, the `last_layer_sigmoid` function should not be used, or a different normalization method needs to be employed.

As we only predicted the mean flow RTT and mean flow rate of flows and mean queue utilization at the interfaces, we can only compare these metrics to findings of related work. Compared to the results of Geyer [4] and Mirza et al. [13], our models performance for predicting the mean flow rate lies somewhere in between. While Mirza et al. report a median absolute relative error of 10 %, our model predicts flow rates for unknown topologies with roughly the same properties with a median absolute relative error of 4 %. Geyer manages to predict the mean flow rate with a median absolute error below 1 %. However, it has to be noted that the topologies being compared here are different. The topologies in the work of Geyer [4] are all daisy-chained switches with servers connected to each switch. In the previous work of Mirza et al. [13] the topology is fixed to a dumbbell topology. The topologies in our approach are all trees, but otherwise not limited in their shape.

The RouteNet model [15] achieves a mean relative error of 2.5 % when predicting the delay in TCP flows with a topology unknown to the model, but comparable in size to the topologies in the training data. This result is comparable to our models' mean relative error of 1.8 % when predicting the mean flow RTTs.

Chapter 7

Conclusion and future work

In this chapter, we summarize the findings of this thesis and try to draw a conclusion in regards to how well GGNNs are suitable for the prediction of TCP performance metrics. A brief overview of possible future work is also presented.

7.1 Conclusion

As shown in Chapter 6, GGNNs prove to be a viable alternative to discrete event network simulation for TCP performance prediction. Once trained, the inference process is magnitudes faster than simulation (see Sections 6.1.1 and 6.2). The ability to present an already trained model with arbitrary shapes of topolgies is what really makes GGNNs attractive for network optimization purposes. However, in order to receive results with desirable accuracy, the graph representation and training data have to be chosen wisely in regards to the metric to be predicted.

The GRs presented in this thesis (Chapter 4) produced mixed results for the three metrics we tried to predict. The RTT prediction was most successful, being able to predict the mean flow RTT with a mean absolute relative error of around 1–2 % in the best case scenario. If trained with a more diverse dataset, e.g. a wider range of queue sizes, these metrics are achieved for the previously worse performing datasets as well (see Section 6.2.3). The flow rate prediction produced overall less acurrate results. Considering that the approach by Geyer [4] also utlizes GGNNs but achieves better accuracy for flow rate predictions, there is potential for further improvement, e.g. by modeling the network graph differently. The third metric, queue utilization at the sending interfaces, was predicted with relatively high accuracy in terms of absolute errors, and thus could also be useful in real world applications. We found that the minmax bounds should be

chosen carefully, especially if the data is passed through a sigmoid function in the end, as this can limit the model's ability to generalize.

The model's ability to generalize to larger networks is somewhat limited by the training data, as it struggles to predict metrics for flows with path lengths that are significantly longer than the ones encountered during training. The same goes for the generalization to different queue sizes, albeit with a less significant impact.

However, as long as the metric to be predicted is not too far off the range of the training data, atleast for the RTT and flow rate predictions, the error might still be in an acceptable range, depending on the application purpose.

7.2 Future work

In the future, the framework could be extended to include a model with an LSTM memory cell as outlined in Section 2.3.2. Geyer has implemented this approach and found that it can improve the accuracy of GGNN models in some cases, so this is definetely an option to explore for future improvement [4]. As no implementation for the PyTorch-Geometric framework exist at the time of writing, we decided to omit the LSTM cell implementation from this thesis, as the implementation would not have been a trivial task.

Another thing that was prepared at the beginning but then later ommitted was the prediction of the flow's loss metric, and the inclusion of lossy links in the simulation. As seen in the example topology JSON file in Listing A.1, the format already includes a `loss` attribute for each link, but those are consistently set to zero by the generator. As we were constrained by time, we decided to focus on the prediction of the presented metrics first before introducing another variable. The implemented simulation program is however fully capable of simulating those potentially lossy links and recognizes the loss attribute from the JSON file. The parsers which produce the GRs would have to be adjusted to include these link loss attributes in order for the network to be able to learn them.

Another topic which could be explored is the inclusion of arbitrary, non tree-like topologies, which allow multiple routes to exist between network nodes. That would require the routing information to be embedded in the topology description and graph representation in some way. Since ns-3 makes it rather difficult to acces the routing tables when populated by the automatic static routing helper, a custom implementation would probably be required.

Finally, the reason for the erroneous behavior of TCP Vegas and TCP BBR in the ns-3 simulation program discussed in Section 5.2 could be investigated. If TCP Vegas and TCP BBR flows would produce reasonable results in the simulator, their effects on networks could hopefully be learned by the GGNN as well.

Chapter A

Appendix

A.1 Example JSON topology file

Listing A.1: Example Topology File generated by topology generator script

```
{
    "node_count": 4,
    "nodes": [
        {
            "id": 0,
            "tcp_congestion_algo": "ns3::TcpCubic",
            "queue_size": 125
        },
        {
            "id": 1,
            "tcp_congestion_algo": "ns3::TcpCubic",
            "queue_size": 125
        },
        {
            "id": 2,
            "tcp_congestion_algo": "ns3::TcpCubic",
            "queue_size": 125
        },
        {
            "id": 3,
            "tcp_congestion_algo": "ns3::TcpCubic",
            "queue_size": 125
        }
    ],
    "link_count": 3,
    "links": [
        {
            "id": 0,
            "lhs": 0,
            "rhs": 1,
            "bandwidth": 80,
            "delay": 14.407,
            "loss": 0
        },
        {
```

```
            "id": 1,
            "lhs": 1,
            "rhs": 2,
            "bandwidth": 30,
            "delay": 35.008,
            "loss": 0
        },
        {
            "id": 2,
            "lhs": 0,
            "rhs": 3,
            "bandwidth": 40,
            "delay": 47.937,
            "loss": 0
        }
    ],
    "flow_count": 1,
    "flows": [
        {
            "id": 0,
            "host_a": 2,
            "host_b": 3,
            "start": 0.0,
            "time_limit": 60.0,
            "data_limit": null,
            "route": [
                2,
                1,
                0,
                3
            ]
        }
    ],
    "sim_duration": 60.0,
    "graphviz": "strict graph links {\nnode [shape=ellipse]; N0; N1; N2; N3;\nN0
        -- N1 [label=\"14.407ms\"];\nN1 -- N2 [label=\"35.008ms\"];\nN0 -- N3 [
        label=\"47.937ms\"];\n}"
}
```

A.2 List of acronyms

IP Internet Protocol
QT Queueing Theory
ML Machine Learning
NN Neural Network
GR Graph Representation
SVR Support Vector Regression
RNN Recurrent Neural Network
GNN Graph Neural Network
GRU Gated Recurrent Unit
GGNN Gated Graph Neural Network
FFNN Feed-Forward Neural Network
LSTM Long Short-Term Memory
TCP Transmission Control Protocol
UDP User Datagram Protocol
JSON JavaScript Object Notation
CSMA Carrier-Sense Multiple Access
PPP Point-to-Point Protocol
RTT Round-Trip Time
CSV Comma-separated values
BBR Bottleneck Bandwidth and Round-trip propagation time, TCP congestion control algorithm
RNG Random Number Generator
LR Learning Rate
MPE Mean Percentage Error
KPI Key Performance Indicator
SDN Software-defined Networking
CPU Core Processing Unit
GPU Graphics Processing Unit
ReLU Rectified Linear Unit

Bibliography

[1] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose, "Modeling TCP Reno performance: A simple model and its empirical validation", *IEEE/ACM transactions on Networking*, vol. 8, no. 2, pp. 133–145, 2000. DOI: https://doi.org/10.1109/90.842137.

[2] *ns-3 Project Website*, https://ns-nam.org/, Accessed: 2022-05-15.

[3] *OMNet++ Project Website*, https://omnetpp.org/, Accessed: 2022-05-15.

[4] F. Geyer, "DeepComNet: Performance evaluation of network topologies using graph-based deep learning", *Perform. Evaluation*, vol. 130, pp. 1–16, 2019. DOI: 10.1016/j.peva.2018.12.003. [Online]. Available: https://doi.org/10.1016/j.peva.2018.12.003.

[5] I. J. Goodfellow, Y. Bengio, and A. C. Courville, *Deep Learning* (Adaptive computation and machine learning). 2016, ISBN: 978-0-262-03561-3. [Online]. Available: http://www.deeplearningbook.org/.

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction* (Adaptive computation and machine learning). 1998, ISBN: 978-0-262-19398-6. [Online]. Available: https://www.worldcat.org/oclc/37293240.

[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model", *IEEE Trans. Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009. DOI: 10.1109/TNN.2008.2005605. [Online]. Available: https://doi.org/10.1109/TNN.2008.2005605.

[8] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN", *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2260–2270, 2020. DOI: 10.1109/JSAC.2020.3000405. [Online]. Available: https://doi.org/10.1109/JSAC.2020.3000405.

[9] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains", in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2, 2005, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

[10] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated Graph Sequence Neural Networks", in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.05493.

[11] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and 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 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, A. Moschitti, B. Pang, and W. Daelemans, Eds., 2014, pp. 1724–1734. DOI: 10.3115/v1/d14-1179. [Online]. Available: https://doi.org/10.3115/v1/d14-1179.

[12] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[13] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A Machine Learning Approach to TCP Throughput Prediction", *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1026–1039, 2010. DOI: 10.1109/TNET.2009.2037812. [Online]. Available: https://doi.org/10.1109/TNET.2009.2037812.

[14] A. Mestres, E. Alarcón, Y. Ji, and A. Cabellos-Aparicio, "Understanding the Modeling of Computer Network Delays using Neural Networks", in *Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks, Big-DAMA@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*, P. Casas, M. Mellia, A. Dainotti, and T. Zseby, Eds., 2018, pp. 46–52. DOI: 10.1145/3229607.3229613. [Online]. Available: https://doi.org/10.1145/3229607.3229613.

[15] K. Rusek, J. Suárez-Varela, A. Mestres, P. Barlet-Ros, and A. Cabellos-Aparicio, "Unveiling the potential of Graph Neural Networks for network modeling and optimization in SDN", in *Proceedings of the 2019 ACM Symposium on SDN Research, SOSR 2019, San Jose, CA, USA, April 3-4, 2019*, 2019, pp. 140–151. DOI: 10.1145/3314148.3314357. [Online]. Available: https://doi.org/10.1145/3314148.3314357.

[16] M. F. Galmés, K. Rusek, J. Suárez-Varela, S. Xiao, X. Cheng, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet-Erlang: A Graph Neural Network for Network Performance Evaluation", *CoRR*, vol. abs/2202.13956, 2022. arXiv: 2202.13956. [Online]. Available: https://arxiv.org/abs/2202.13956.

[17] *GraphViz Project Website*, https://graphviz.org/, Accessed: 2022-05-15.

[18] *GNU Parallel Project Website*, https://www.gnu.org/software/parallel/, Accessed: 2022-05-15.

[19] *Neural Network Intelligence Project Website*, https://www.microsoft.com/en-us/research/project/neural-network-intelligence/, Accessed: 2022-05-15.

[20] X. Bresson and T. Laurent, "Residual Gated Graph ConvNets", *CoRR*, vol. abs/1711.07553, 2017. arXiv: 1711.07553. [Online]. Available: http://arxiv.org/abs/1711.07553.