GNNet Challenge 2021 report (1st place)

Bruno Klaus de Aquino Afonso*

Representing PaRaNA (Pattern Recognition and Network Analysis) group

Institute of Science and Technology, Federal University of São Paulo, Brazil

October 13, 2021

Introduction

During the year of 2021, the International Telecommunication Union (ITU) once again brought to the forefront the use of machine learning as a means to maximize the efficiency of 5G. The 2nd edition of the *"ITU AI/ML in 5G challenge"*[1], open to participants from all over the world, introduced a diverse set of challenges related to the development and training of machine learning models to solve particular problems within the realm of 5G networks. This report details the 1st place solution of the challenge proposed by the Barcelona Neural Networking Center, named *Graph Neural Networking Challenge 2021 - Creating a Scalable Digital Network Twin*, a.k.a. **GNNet Challenge 2021**.

Much like in the previous year, the GNNet challenge was centered around the task of network modeling: given a network topology and routing configuration, one must predict performance metrics such as the per-path-delay. Moreover, the GNNet Challenge 2021 was proposed with a specific goal in mind: to address the current limitations of GNN architectures, whose generalization suffers greatly when predicting on larger graphs. This was verified by the organizers to be the case for *RouteNet*[2], a message-passing GNN model that influenced most solutions put forth in the 2020 edition of the challenge.

To understand the solution detailed in this report, it is helpful to look at the previous approaches (Table 1). Packet simulators were not allowed in the competition in principle due to excessive running times; analytical approaches generalize well and run fast, but they do not offer competitive performance; RouteNet is still fast and more performant than analytical approaches, but fails to generalize to larger graphs. For our proposed solution, we **extract invariant features from the analytical approach**, and feed them to a GNN. This way, we can **maintain generalization while outperforming the purely analytical** approach.

	Fast Enough?	**Has top tier performance?**	**Generalizes to larger graphs?**
Analytical	✓	✗	✓
Packet simulators	✗(prohibited)	✓	✓
RouteNet	✓	✓	✗
Proposed solution	✓	✓	✓

Table 1: Types of approaches

*email: bruno.klaus@unifesp.br

Framework

Our model was built from scratch in the Python language using Pytorch 1.8.1 [3] and Pytorch Geometric 1.7.0 [4]. The code requires a GPU; we used an RTX 3080 with 10GB VRAM. Moreover, 16GB of RAM is enough to not run out of memory. Our code is divided into 3 different Jupyter notebooks: one for creating the dataset, two for **2 similar models** whose **average** constituted the final prediction used for this challenge.

Input format In order to improve speed when training our model, we modify the script provided to challenge participants [1]. Our script allows one to run multiple processes in parallel, to speed up the creation of this "converted" dataset.

	Path	**Number of Files**
Training	./dataset/converted_train	120000
Validation	./dataset/converted_validation	3120
Test	./dataset/converted_test	1560

Table 2: Converted dataset information

Due to the large amount of files, we recommend an SSD with at least 60GB of memory available. There are three types of attribute: path, link and global attributes. As noted in Table 3, there are five path attributes. We zero out

	Parameter type	**Note**
`p_TotalPktsGen`	Path	Unused (multiplied by 0)
`p_PktsGen`	Path	
`p_AvgBw`	Path	Divided by 1000
`p_EqLambda`	Path (distribution)	Divided by 1000
`p_AvgPktsLambda`	Path (distribution)	Divides all other attributes

Table 3: Path attributes

`p_TotalPktsGen`, as we found the inclusion of simulation time not helpful. In order to subject the network to **more varied input values**, we divide link capacity `l_LinkCapacity` (the sole link attribute) and all path attributes by `p_AvgPktsLambda`. We also extracted 4 global attributes, but ended up not using them (i.e. zeroing them out). We do not use any attributes related to flow performance measurements or port statistics as input to our models.

Our GNN model works with a matrix $\boldsymbol{X}$, which is the concatenation of all attributes. The number of rows is equal to the sum of the number of paths, links and nodes. These columns all remain fixed, so we also need to add "hidden state" columns to each of these entities. These columns are used to perform message-passing, taking in information about other "hidden state" columns and also fixed columns. We denote by $\boldsymbol{X}_P, \boldsymbol{X}_L, \boldsymbol{X}_N$ the fixed columns of paths,

[1] https://github.com/BNN-UPC/GNNetworkingChallenge/tree/2021_Routenet_TF

links, and nodes. The hidden state columns are zero-initialized and denoted by $\boldsymbol{X}_{Ph}, \boldsymbol{X}_{Lh}, \boldsymbol{X}_{Nh}$. We intended to put global attributes into $\boldsymbol{X}_N$ but eventually opted for setting them to zero. We standardize all features before feeding them to the model.

Next, we must go over network topology. We denote the topology by $\boldsymbol{E}$. The conditions for the existence of edges are:

- $\boldsymbol{E}_{PL}$: Whenever a link is part of a path
- $\boldsymbol{E}_{PN}$: Whenever a node is part of a path
- $\boldsymbol{E}_{LN}$: Whenever a node is part of a link

In practice, edges are directed. We may use the terminology $\boldsymbol{E}_{PL}$ to indicate that the direction is *path-to-link*, whereas $\boldsymbol{E}_{LP}$ is *link-to-path*. In addition, our code contains a special function, `SeparateEdgeTimeSteps`, which is able to output a list separating $\boldsymbol{E}_{LP}$. The k-th element of this list has all of the $\boldsymbol{E}_{LP}$ edges satisfying a condition: that the link is the k-th one found while traversing the path. This separation allows us to preserve order information and use recurrent layers.

Message Passing models There are two message passing models, which are listed as Algorithm 1 and Algorithm 2 **(see appendix)**. The main difference is that the first model includes nodes in the message mechanism, which are ignored by the second model. For both models, we feed the initial input matrix $\boldsymbol{X}$ to a **multilayer perceptron (MLP)**. Then, we perform a number of message-passing iterations using convolutions. First the path entities receive messages, then nodes, and lastly links.

Messages are exchanged from links to paths using a single **Chebyshev Graph Convolutional Gated Recurrent Unit Cell** [5] layer imported from Pytorch Geometric Temporal [6]. All other convolutions were set to be **Graph Attention (GAT)** [7] layers, and different GAT layers are used for each iteration. We set the first few hidden columns to be equal to the baseline features, so that this information is preserved similarly to the other fixed features. After the message-passing rounds, we feed $\boldsymbol{X}_L$ and $\boldsymbol{X}_{Lh}$ to another MLP to obtain the prediction for *average queue utilization*. Finally, the average path delay is obtained using the formula

$$\texttt{pathDelay} \approx \sum_{i=0}^{\texttt{n\_links}} \texttt{delayLink}(i) \tag{1}$$

where

$$\texttt{delayLink}(i) = \texttt{avg\_utilization}_i \times (\texttt{queue\_size}_i / \texttt{link\_capacity}_i) \tag{2}$$

Baseline Perhaps the most important aspect of our model is its use of an analytical baseline that serves as a feature extraction step. This algorithm (Algorithm 3) is based on *Queueing Theory (QT)*, and iteratively calculates the

traffic on links and blocking probabilities. After these iterations, we calculate the traffic intensity ρ, probability of being in state zero π_0, and predicted average occupancy $\boldsymbol{L}$. The first two are used as features only in the 2nd model. In addition, we also extract the baseline's per-path-delay prediction, using Equation (1).

Hyperparameters The other significant difference between the two models lies in the model size. When developing the 2nd model, in addition to throwing away the node entities, we opted to scale down as much as possible. This can be observed by looking at the size (i.e. number of columns) of $\boldsymbol{X}_{Lh}, \boldsymbol{X}_{Ph}, \boldsymbol{X}_{Nh}$, as well as the hidden layer size for the second multilayer perceptron.

	# of hidden input columns	**MLP_1**	**MLP_2**
Model 1	$\boldsymbol{X}_{Ph}$:64 $\boldsymbol{X}_{Lh}$:64 $\boldsymbol{X}_{Nh}$:64	Seq(Linear(128), LeakyRELU(), Linear(inp_dim), LeakyRELU())	Seq(Linear(512), LeakyRELU(), Linear(512), LeakyRELU(), Linear(1))
Model 2	$\boldsymbol{X}_{Ph}$:8 $\boldsymbol{X}_{Lh}$:8	Seq(Linear(128), LeakyRELU(), Linear(inp_dim), LeakyRELU())	Seq(Linear(128), LeakyRELU(), Linear(32), LeakyRELU(), Linear(1))

The number of message passing rounds for both models is 3. Model 1 uses 5 baseline iterations, whilst Model 2 reduces that to 3 iterations.

	Baseline iterations	**Message passing iterations**
Model 1	5	3
Model 2	3	3

Lastly, we (mistakenly) set the initial guess of blocking probabilities to 0.3 when training our models. This was made apparent only while writing this report. It is unknown whether setting them to zero produces better results.

Training and results We use the Adam optimizer with learning rate equal to `1e-03`. The batch size is set to 16. Even when setting a seed, there seemed to be some small variation between runs, this could be due to GPU operations.

To perform early stopping, we evaluate, after each epoch, on a small subset of each of the 3 validation sets. Each epoch corresponds to going through some random 10% of the training set; in addition to this, we also select only a constant subset of each validation set, sacrificing some accuracy in order to speed up the process.

After each epoch, validation stats are printed and a new model file is saved to the `./model` folder. We submitted a few models from different epochs. In

particular, it seemed that validation set 1 overestimated the MAPE metric on the test set, going as high as 3, whereas validation sets 2 and 3 were more in line with what one would expect. Submissions that prioritized losses on validation sets 2 and 3 were usually more successful (unless the MAPE on validation 1 was significantly large).

Training on Model 1 took just over 8 hours. Training on model 2 takes just over an hour. Respective model weights are saved as **22_setembro_modelo.pt** and **29_setembro_modelo.pt**. While compiling this report, we loaded the model weights and confirmed that they indeed produce the same submissions sent to the challenge.

Tensorboard support was added to the code a few days after training Model 1. It provides another way to look at the performance metrics on-the-fly.

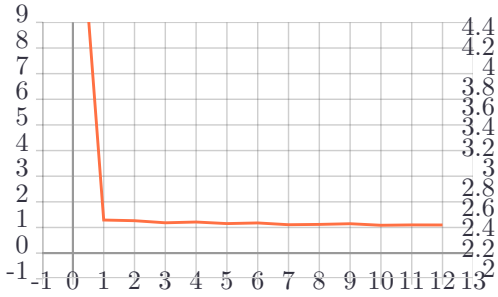

Figure 1: Training set

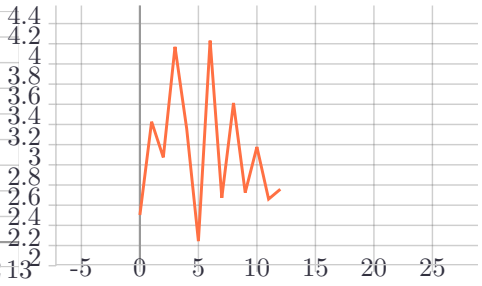

Figure 2: Validation set #1

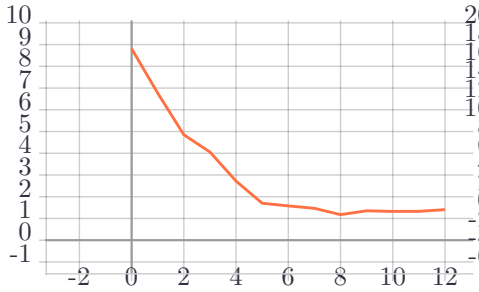

Figure 3: Validation set #2

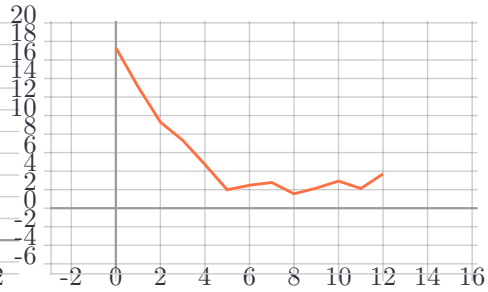

Figure 4: Validation set #3

Figure 5: Model 2 tensorboard run for the training set and validation subsets. The submission used for the final prediction was the one for epoch #8. Note that the validation data is the same for each step, while a random 10% of the training set is evaluated during an epoch.

The obtained results are listed in Table 4, which lists the MAPE for the validations subsets 1/2/3, as well as for the final test set. Our first submission to the challenge was a version of Model 1 without using baseline features, and it attained a MAPE of 22.58 on the test set. Anecdotally, we observed much

higher MAPE than that if we did not divide features by `p_AvgPktsLambda`.

	Val. 1	**Val. 2**	**Val. 3**	**Test**
Model 1 (Sep 22nd)	2.71	1.33	1.65	1.45
Model 2 (Sep 29th)	3.61	1.17	1.55	1.45
(Model 1+ Model 2)/2	—	—	—	1.27
Baseline	12.10	9.18	9.51	?
Model 1 w/o baseline	—	—	—	22.58

Table 4: MAPE error for validation and test sets

The final versions of both models performed almost identically on the test set. On the validation sets, Model 1 was better than Model 2 on validation set 1, and worse on validation sets 2 and 3. Their average was able to attain the lowest MAPE of 1.27. Even though the rules of this challenge allowed up to 20 submissions, we only used around 10. We planned on averaging runs from Model 2, but a problem with the evaluation website left us with only a single submission to make. This made us opt for the safer option of averaging Model 1 and Model 2.

References

[1] José Suárez-Varela et al. The graph neural networking challenge: a worldwide competition for education in ai/ml for networks. *ACM SIGCOMM Computer Communication Review*, 51(3):9–16, 2021.

[2] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert 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*, pages 140–151, 2019.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[4] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[5] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent net-

works. In *International Conference on Neural Information Processing*, pages 362–373. Springer, 2018.

[6] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, , Guzman Lopez, Nicolas Collignon, and Rik Sarkar. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, 2021.

[7] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Appendix

Algorithm 1 Model 1 (submitted on September 22nd)

Require: $\boldsymbol{X}$ = `Concatenate([`$\boldsymbol{X}_P, \boldsymbol{X}_{Ph}, \boldsymbol{X}_L, \boldsymbol{X}_{Lh}, \boldsymbol{X}_N, \boldsymbol{X}_{Nh}$`],axis=1)`
Require: `baseline_path`, `baseline_link`: baseline predictions
Require: $\boldsymbol{E}$: network topology
Require: `NUM_iterations`: number of message-passing iterations
  `E_lp_list` $\leftarrow$ `SeparateEdgeTimeSteps`$(\boldsymbol{E}_{LP})$
  $\boldsymbol{X} \leftarrow$ `MLP_1`$(\boldsymbol{X}, \boldsymbol{E}_{LN})$
  for $0 \leq i <$ `NUM_ITERATIONS` **do**
                                                ▷ Paths receive messages
    $\boldsymbol{X}_{Ph} \leftarrow$ `LeakyRELU`$(\text{Conv}_{i,\texttt{node\_to\_path}}(\boldsymbol{X}, \boldsymbol{E}_{NP}))$
    $H \leftarrow$ `None`
    for $0 \leq k <$ `E_lp_list`$.length$ **do**
      $H \leftarrow (\text{GConvGRU}_{0,\texttt{link\_to\_path}}(\boldsymbol{X}, H, \texttt{E\_lp\_list[k]}))$
    end for
    $\boldsymbol{X}_{Ph} \leftarrow$ `LeakyRELU(H/(E_lp_list.length))`
    $(\boldsymbol{X}_{Ph})$`[:, 0:baseline_path.shape[1]]` $\leftarrow$ `baseline_path`
                                                ▷ Nodes receive messages
    $\boldsymbol{X}_{Nh} \leftarrow$ `LeakyRELU`$(\text{Conv}_{i,\texttt{path\_to\_node}}(\boldsymbol{X}, \boldsymbol{E}_{PN}))$
    $\boldsymbol{X}_{Nh} \leftarrow \boldsymbol{X}_{Nh} +$ `LeakyRELU`$(\text{Conv}_{i,\texttt{link\_to\_node}}(\boldsymbol{X}, \boldsymbol{E}_{LN}))$
                                                ▷ Links receive messages
    $\boldsymbol{X}_{Lh} \leftarrow$ `LeakyRELU`$(\text{Conv}_{i,\texttt{node\_to\_link}}(\boldsymbol{X}, \boldsymbol{E}_{NL}))$
    $\boldsymbol{X}_{Lh} \leftarrow$ `LeakyRELU`$(\text{Conv}_{i,\texttt{path\_to\_link}}(\boldsymbol{X}, \boldsymbol{E}_{PL}))$
    $(\boldsymbol{X}_{Lh})$`[:, 0:baseline_link.shape[1]]` $\leftarrow$ `baseline_link`
  end for
  $\boldsymbol{L} \leftarrow$ `Concatenate`(X_L, X_{Lh})
  $\boldsymbol{L} \leftarrow$ `Sigmoid`(`MLP_2`$(\boldsymbol{L}))$ ▷ Predicts average queue utilization
  return `GetPathDelay`$(\boldsymbol{L}, \boldsymbol{E}_{LP})$ ▷ Obtains per-path-delay

Algorithm 2 Model 2 (submitted on September 29th)

Require: $\boldsymbol{X}$ = `Concatenate([`$\boldsymbol{X}_P, \boldsymbol{X}_{Ph}, \boldsymbol{X}_L, \boldsymbol{X}_{Lh}, \boldsymbol{X}_N, \boldsymbol{X}_{Nh}$`],axis=1)`
Require: `baseline_path`, `baseline_link`: baseline predictions
Require: $\boldsymbol{E}$: network topology
Require: `NUM_iterations`: number of message-passing iterations
  `E_lp_list` $\leftarrow$ `SeparateEdgeTimeSteps`$(\boldsymbol{E}_{LP})$
  $\boldsymbol{X} \leftarrow$ `MLP_1`$(\boldsymbol{X}, \boldsymbol{E}_{LN})$
  for $0 \leq i <$ `NUM_ITERATIONS` **do**
                                                ▷ Paths receive messages
    $H \leftarrow$ `None`
    for $0 \leq k <$ `E_lp_list`$.length$ **do**
      $H \leftarrow$ (`GConvGRU`$_{0,\texttt{link\_to\_path}}(\boldsymbol{X}, H,$ `E_lp_list[k]`$)$)
    end for
    $\boldsymbol{X}_{Ph} \leftarrow$ `LeakyRELU(H/(E_lp_list.length))`
    $(\boldsymbol{X}_{Ph})$`[:, 0:baseline_path.shape[1]]` $\leftarrow$ `baseline_path`
                                                ▷ Links receive messages
    $\boldsymbol{X}_{Lh} \leftarrow$ `LeakyRELU(`$\texttt{Conv}_{i,\texttt{path\_to\_link}}(\boldsymbol{X}, \boldsymbol{E}_{PL})$`)`
    $(\boldsymbol{X}_{Lh})$`[:, 0:baseline_link.shape[1]]` $\leftarrow$ `baseline_link`
  end for
  $\boldsymbol{L} \leftarrow$ `Concatenate`(X_L, X_{Lh})
  $\boldsymbol{L} \leftarrow$ `Sigmoid(MLP_2(`$\boldsymbol{L}$`))` ▷ Predicts average queue utilization
  return `GetPathDelay`$(\boldsymbol{L}, \boldsymbol{E}_{LP})$ ▷ Obtains per-path-delay

Algorithm 3 Baseline

Require: $\boldsymbol{E}$: network topology
Require: `p_PktsGen`: Packets generated per time unit for each path
Require: `l_LinkCapacity`: Capacity of each link
Require: `NUM_iterations`: number of iterations

Initialize PB as a vector with constant values for each link.
$B \leftarrow 32$
Let $\lambda_{k,i}$ be the amount of traffic from path k passing through link i.
$\lambda_{k,k(i)}$ is the traffic from path k passing through its i-th edge.
for $0 \leq it <$ `NUM_iterations` **do**
  $A \leftarrow$ `p_PktsGen` ▷ A is the demand on each path
  for each path k **do**
    Let m_k be the max number of edges in path k.
    $\lambda_{k,k(1)} \leftarrow A_k$
    $\forall j \in \{1.. \leq m_k\} : \lambda_{k,k(j)} \leftarrow A_k \prod_{i=1}^{j-1} PB_i$
  end for
  for each link l **do** ▷ Get total traffic on links
    $T_l \leftarrow \sum_{\exists i: l=k(i)} \lambda_{k,l}$
    $\rho_l \leftarrow T_l / \texttt{l\_LinkCapacity}_l$
    $\texttt{PB}_l \leftarrow \frac{(1-\rho_l)\rho_l^B}{(1-\rho_l)^{B+1}}$ ▷ Update blocking probabilities
  end for
end for
$\pi_0 \leftarrow (1-\rho)/(1-\texttt{pow}(\rho, B+1))$ ▷ Prob. that the queue is at state 0
$\boldsymbol{L} \leftarrow \frac{1}{B}(\pi_0 + \sum_{j=1}^{B} j(\pi_0 \cdot \texttt{pow}(\rho, j)))$
`baseline_link` $\leftarrow [\pi_0, \rho, \boldsymbol{L}]$ ▷ Obs: For the 1st model, we take only $\boldsymbol{L}$
`baseline_path` $\leftarrow \texttt{GetPathDelay}(\boldsymbol{L}, \boldsymbol{E}_{LP})$
return `baseline_link`, `baseline_path`