Graph Neural Networking Challenge 2020 - Steredeg's solution

LOÏCK BONNIOT, InterDigital and Univ Rennes, Inria, CNRS, IRISA
CHRISTOPH NEUMANN, InterDigital
FRANÇOIS SCHNITZLER, InterDigital
FRANÇOIS TAÏANI, Univ Rennes, Inria, CNRS, IRISA

This document describes the winning solution to the GNN Challenge 2020 organized by the Barcelona Neural Networking Center for the ITU Artificial Intelligence/Machine Learning in 5G Challenge. We first describe our methodology, then give the set of hyper-parameters that allowed us to achieve the best score with an average relative error of 1.53%, and finally explain and discuss our results.

1 INTRODUCTION

Software Defined Networking (SDN) is the concept of programmatically defining and (re)configuring a network from a centralized control plane. The great level of flexibility introduced by SDN has been embraced by many players, *e.g.* in 5G networks, in datacenters or in broadband ISP networks, and SDN is now effectively controlling many of these networks. However, communication networks are complex distributed systems in which every configuration parameter, the topology and the capacity of the network itself can have an important impact on the network's performance (e.g. resulting jitter, latency, loss, *etc.*). For example, the routing policy and the priority policies can largely affect the resulting latency of the concerned traffic flows. Thus, it becomes crucial to have powerful network modeling tools that allow to predict the impact of a network configuration (which may include a change of the network topolgy) on a network's perfomance before applying it.

Machine learning arises as a promising solution to build accurate predictive models able to operate on networks in real time. Actually, the ITU highlights relevant use-cases for machine learning based network configuration in order to optimize the Quality of Experience (QoE) [2]. In particular, one requirement is that machine learning models should be able to take as input parameters such as the network's bandwidth, bit rate, cache size and network states and return a predicted QoE which includes *e.g.* latency, jitter.

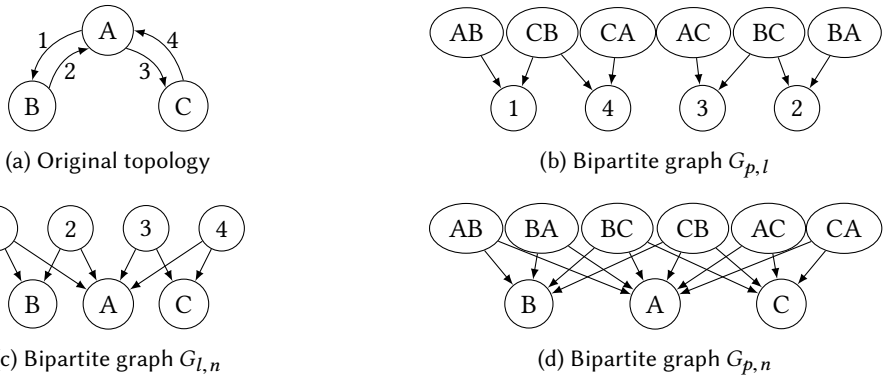
Graph Neural Networks (GNN) seem to be particularly promising as they may allow to capture the communication network topology and to operate and predict on network topologies never encountered before, as showed by Rusek et al. [4]. In this context, the Barcelona Neural Networking Center organized the GNN Challenge 2020 as part of the AI/ML in 5G Challenge of the ITU.

This document describes our winning solution to the GNN Challenge 2020. We first describe our methodology, then give the set of hyper-parameters that allowed us to achieve the best score with an average relative error of 1.53%, and finally explain and discuss our results. The source code of the solution is freely available on demand.[1]

2 MODEL OVERVIEW

Our model is based on novel message-passing convolutional layers on graph neural networks of the form $\mathbf{x}'_i = \gamma\left(\mathbf{x}_i, \square_{j\in\mathcal{N}(i)}\,\phi\left(\mathbf{x}_i, \mathbf{x}_j\right)\right)$, where $\mathbf{x}_i$ is the hidden state of node i; γ and ϕ are differentiable functions; $\square$ is an aggregation function over the neighbors j of i. We rely on the methods for message passing on graphs introduced by [1] and extend with specific types of convolutions (see section 2.2). We use the base class provided by Pytorch Geometric [3].

[1]Please contact the second author via email (firstname.lastname@interdigital.com) to request access to the source code.



(a) Original topology

(b) Bipartite graph $G_{p,l}$

(c) Bipartite graph $G_{l,n}$

(d) Bipartite graph $G_{p,n}$

Fig. 1. Example of a basic topology leading to the three presented bipartite graphs.
Nodes are denoted by a single letter, links by a number, paths by two letters (XY meaning path from X to Y).

2.1 From network topology to bipartite graphs

We first define $\mathbf{x}$ and $\mathcal{N}$ for the GNNet problem. Each sample contains the network topology, features for paths (F_p, for each source-destination pair), links (F_l) and nodes (F_n), and finally routing for each path. For every sample, we construct $G_{p,l}$, $G_{l,n}$ and $G_{p,n}$, three bipartite graphs where paths, nodes and links are represented as vertices and edges are undirected.

- $G_{p,l}$ connects each path to the links it uses
- $G_{l,n}$ connects each link to the two nodes that have a topological link between them
- And $G_{p,n}$ that directly connects each path to the nodes it traverses.

Figure 1 presents an example of the construction of the three aforementioned bipartite graphs from a 3-nodes simple network topology. We define $\mathcal{N}_{A,B}(i)$ as the indices j of vertices where (j,i) is an edge in $G_{A,B}$. Conversely, we define $\tilde{\mathcal{N}}_{A,B}(i)$ as the indices j of vertices where (i,j) is an edge in $G_{A,B}$. We define the hidden states for paths, links and nodes as $\mathbf{p}$, $\mathbf{l}$, $\mathbf{n}$ respectively. The hidden states are initialized to the features of each path, link and node respectively. In our implementation, all the hidden states are stored in a single $\mathbf{x}$ matrix for compliance with Pytorch Geometric APIs. This requires complex index handling, so we use separate notations in this document.

2.2 Aggregations in graph message-passing convolutions

In our model, we use two different functions for $\square$: a set of permutation-invariant functions Ω when the order of inputs should be ignored, and a Recurrent Neural Network (RNN) otherwise. In both cases, ϕ is a fully-connected layer with an activation function that only depends on $\mathbf{x}_j$. (We denote the concatenation operator by $\|$.)

$$\Omega : \mathcal{X} = \{\phi(\mathbf{x}_a), \ldots, \phi(\mathbf{x}_z)\} \mapsto (\text{avg}(\mathcal{X}) \parallel \text{sum}(\mathcal{X}) \parallel \min(\mathcal{X}) \parallel \max(\mathcal{X}) \parallel \text{var}(\mathcal{X}))$$
$$RNN : \mathcal{X} = \{\phi(\mathbf{x}_a), \ldots, \phi(\mathbf{x}_z)\} \mapsto GRU(\mathcal{X})$$

2.3 Final model

Using the definitions from the two previous subsections, we can now describe our Graph Neural Network model at a high level. As illustrated in Figure 2 and detailed in algorithm 1, the model first initializes hidden states $\mathbf{p}, \mathbf{l}, \mathbf{n}$ using available features for every path, link and node (line 1). Then, it successively updates the hidden states using the message-passing convolution over the bipartite graphs defined in subsection 2.1. We repeat the update procedure T times to allow fixed-point convergence of hidden states given the inter-dependencies between paths,

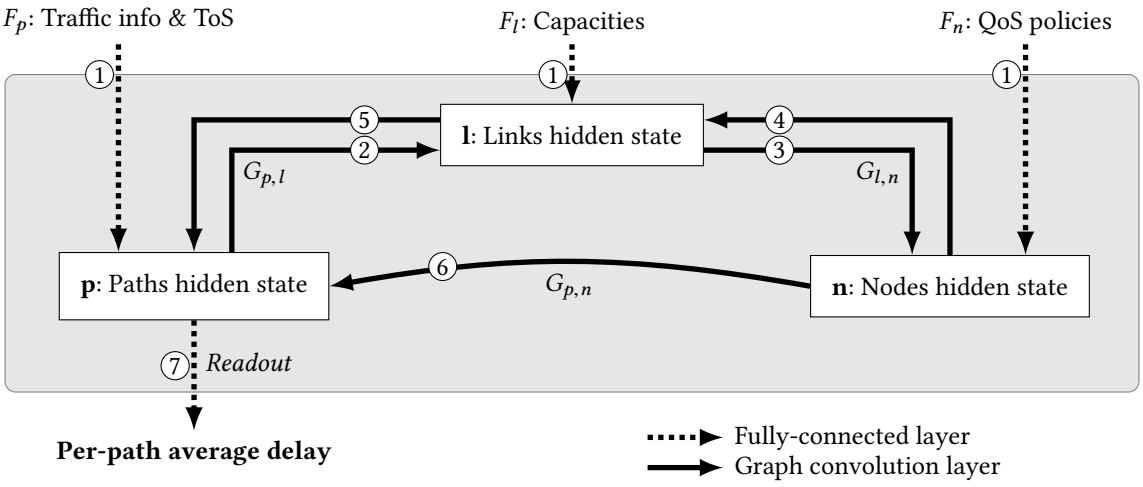


Fig. 2. Visual representation of our GNN model. Firstly, hidden states are initialized from available features in step ①. Using convolution operations over the bipartite graphs $G_{p,l}$ and $G_{l,n}$, hidden states are successively updated through steps ② to ⑤ (repeated T times). Paths' hidden states are finally updated using a convolution over $G_{p,n}$ ⑥ and fully-connected layers are used to readout the predicted average delay for each path ⑦.

links and nodes (lines 2 to 7). This method of state convergence was first proposed in Routenet [4] and we also noted its capability to increase model's accuracy. A final convolution is applied to build an additional relation between nodes and paths (line 8) Finally, each path's hidden state is transformed to the final per-path average delay through a series of non-linear fully-connected layers (the "Readout" step from line 9). It is worth noting that the full workflow is applied both during *training* and *inference*.

Trainable parameters are *not* shared between layers (we omit indices for $\Omega, RNN, \gamma, \phi$ to improve readability in algorithm 1). We use RNN aggregation when an *ordering* of vertices is available: for instance links and nodes are ordered within each path (convolutions in steps ⑤ and ⑥ or lines 6 and 8) but there is no ordering relation between links and nodes. Fully connected layers are denoted by *FC*. The actual implementation uses additional padding and concatenation operations to "glue" the layers together.

3 TRAINING

3.1 Features preprocessing

Input features exhibit large differences between them: for instance, paths "EqLambda" vary from 40 to 2000 while links "bandwidth" vary from 10,000 to 100,000. Hence, we standardize continuous features by removing the mean and scaling to unit variance. (This simple preprocessing allowed us to greatly improve on the RouteNet [4] baseline.) Nodes' features only have a limited set of values: there are only 3 possible queuing policies (strict priority SP, weighted fair queuing WFQ, deficit round robin DDR) and 5 combinations of weights for WFQ and DDR, leading to a total of 11 possible combinations. We encode these 11 combinations to a 4-dimensional embedding. We apply another embedding for the paths ToS, transforming the 3 possible values in a dimension of size 4. (Embeddings are also learned during the training.)

Algorithm 1: High-level pseudo-code of our GNN proposal

▷ *State initialization, padding with zeroes*

$\forall i \in \mathbf{p} : \mathbf{p}_i \leftarrow \left[F_{p,i}, 0, \ldots, 0\right], \forall i \in \mathbf{l} : \mathbf{l}_i \leftarrow \left[F_{l,i}, 0, \ldots, 0\right], \forall i \in \mathbf{n} : \mathbf{n}_i \leftarrow \left[F_{n,i}, 0, \ldots, 0\right]$

for $t \leftarrow 1$ **to** T **do**

- $\forall i \in \mathbf{l} : \mathbf{l}_i \leftarrow \gamma\left(\mathbf{l}_i, \Omega_{j \in \mathcal{N}_{p,l}(i)}\, \phi\left(\mathbf{p}_j\right)\right)$ ▷ *Update link hidden state from path hidden state*
- $\forall i \in \mathbf{n} : \mathbf{n}_i \leftarrow \gamma\left(\mathbf{n}_i, \Omega_{j \in \mathcal{N}_{l,n}(i)}\, \phi\left(\mathbf{l}_j\right)\right)$ ▷ *Update node hidden state from link hidden state*
- $\forall i \in \mathbf{l} : \mathbf{l}_i \leftarrow \gamma\left(\mathbf{l}_i, \Omega_{j \in \tilde{\mathcal{N}}_{l,n}(i)}\, \phi\left(\mathbf{n}_j\right)\right)$ ▷ *Update link hidden state from node hidden state ("backward")*
- $\forall i \in \mathbf{p} : \mathbf{p}'_i \leftarrow \gamma\left(\mathbf{p}_i, \Omega_{j \in \tilde{\mathcal{N}}_{p,l}(i)}\, \phi\left(\mathbf{l}_j\right)\right), \mathbf{p}''_i \leftarrow \gamma\left(\mathbf{p}_i, RNN_{j \in \tilde{\mathcal{N}}_{p,l}(i)}\, \phi\left(\mathbf{l}_j\right)\right)$
- $\mathbf{p} \leftarrow FC_0\left(\mathbf{p}' \parallel \mathbf{p}''\right)$ ▷ *Update path hidden state from link hidden state with two aggregation methods*

$\forall i \in \mathbf{p} : \mathbf{p}'_i \leftarrow \gamma\left(\mathbf{p}_i, RNN_{j \in \tilde{\mathcal{N}}_{p,n}(i)}\, \phi\left(\mathbf{n}_j\right)\right)$ ▷ *Bonus convolution from node to path hidden states*

$\mathbf{p} \leftarrow \left(\mathbf{p} \parallel \mathbf{p}' \parallel F_p\right)$ ▷ *Start readout from path hidden states and include features back*

$\mathbf{r}_1 \leftarrow \text{LeakyRELU}\left(FC_1\left(\mathbf{p}\right)\right)$

$\mathbf{r}_2 \leftarrow \text{LeakyRELU}\left(FC_2\left(\mathbf{r}_1\right)\right)$

$\mathbf{r}_3 \leftarrow FC_3\left(\mathbf{r}_2\right)$

return $FC_4\left(\text{LeakyRELU}\left(\mathbf{r}_3\right) \parallel \text{TanH}\left(\mathbf{r}_3\right)\right)$

3.2 Using the logarithm in the loss function

The challenge uses the Mean Average Prediction Error (MAPE) to evaluate solutions against expected average delays. This metric tends to favour small predictions: predicting 1 when the expected result is 5 leads to an error of 80%, while predicting 5 when the expected result is 1 leads to an error of 400%. It seemed important to train our model on this loss function, but this implies some practical considerations. First, it is not possible to normalize or standardize the expected model output (a zero value would lead to a division by zero and negative values do not make sense for MAPE). Then, expected delays range from 0.0075 to 20, with most values staying below 1. To improve numerical stability (and since we are interested in *relative* differences in this challenge) we train over the logarithm of the expected average delays distribution: $\forall d_i \in]0, 20] : d'_i = \log(d_i + 1) > 0$.

4 EVALUATION

We tried many combinations of hyper-parameters for the challenge. In this section, we present the results for the best model (as measured from the provided validation dataset).

4.1 Challenge score

Using a single model, we obtained a MAPE of 1.66% on the evaluation dataset after 750'000 training samples (submission 16, Table 1). We first trained this model on 500'000 samples with a cyclic learning rate scheduler applied to the Adam optimizer (Scheduler1), then relaunched the training from the last trained parameters for 250'000 samples, with a slightly modified learning rate scheduler (Scheduler2).

To further improve our score, we trained *several* models with different hyper-parameters. As a result, our submission number 18 (MAPE of 1.53%) is the *harmonic mean* of 4 models' outputs (Table 2). The advantage of this power mean is that it favours small values, just like the MAPE, allowing to reduce the error in our delay predictions.

Table 1. Hyper-parameters used for submission 16.

Loss function	MAPE over d'_i
Batch size	8 samples
Optimizer	`Adam` with learning rate = 1×10^{-2}, weight decay = 0
Scheduler1	`CyclicLR` with exp decay of 0.999, base learning rate = 1×10^{-8}, max = 1×10^{-3} step every 128 samples, 1280 samples up, 102 400 samples down
Scheduler2	`CyclicLR` with exp decay of 0.999, base learning rate = 1×10^{-10}, max = 1×10^{-4} step every 128 samples, 1280 samples up, 102 400 samples down
Hidden state size	400 for paths, links and nodes (**p**, **l**, **n**)
RNN size	same as hidden state (400)
FC_1 size	512
FC_2 size	256
FC_3 size	256
T	3
Total params	11 465 185

Table 2. Summary of hyper-parameters used in models for submission 18.

Model	Loss function	Hidden state size	FC_1 size	FC_2 size	FC_3 size	T
0	MAPE	400	512	256	256	3
1	MSE	400	512	256	256	3
2	MAPE	300	128	128	256	3
3	MSE	300	128	128	256	3

4.2 Visualization of paths' hidden states

Each path's predicted delay is extracted from the corresponding path's hidden state using the *Readout* fully-connected layers. It is thus interesting to analyze the hidden states to verify the model's ability to extract relevant features to our problem. To visualize the hidden states in an understandable way, we project the high-dimensional hidden states to a 2-dimensional space using t-distributed Stochastic Neighbor Embedding (t-SNE).

Figure 3 shows the result of this projection, with different coloring for relevant features in the case of the delay-prediction problem (each point represents a path). In the top left plot, we color the paths according to the *expected* average delay and we can clearly see a smooth color gradient between low and high latencies, hinting that the hidden states hold sufficient information to give good delay predictions. The top right plot hints that paths with high delays are correlated with paths having a ToS of 2. This is not surprising: this ToS has the lowest weighting for WFQ and DRR QoS policies in the provided datasets. Bottom plots show that the hidden states also contain information about *links parts of each path*. For instance, we can clearly see that paths are grouped by their *lenghts* in the left plot and by the average of their links' bandwidths in the right plot, hinting that convolutional operators are very useful to extract important information.

4.3 Impact of loss function and learning rate policy

As explained in subsection 3.2, our best model was trained over the MAPE from the *logarithm* of the expected average delays. Figure 4 shows the difference in the predicted delays between a model trained with the original delays (left) and our best model trained with logarithmic delays (right). For high expected delays (> 10), we

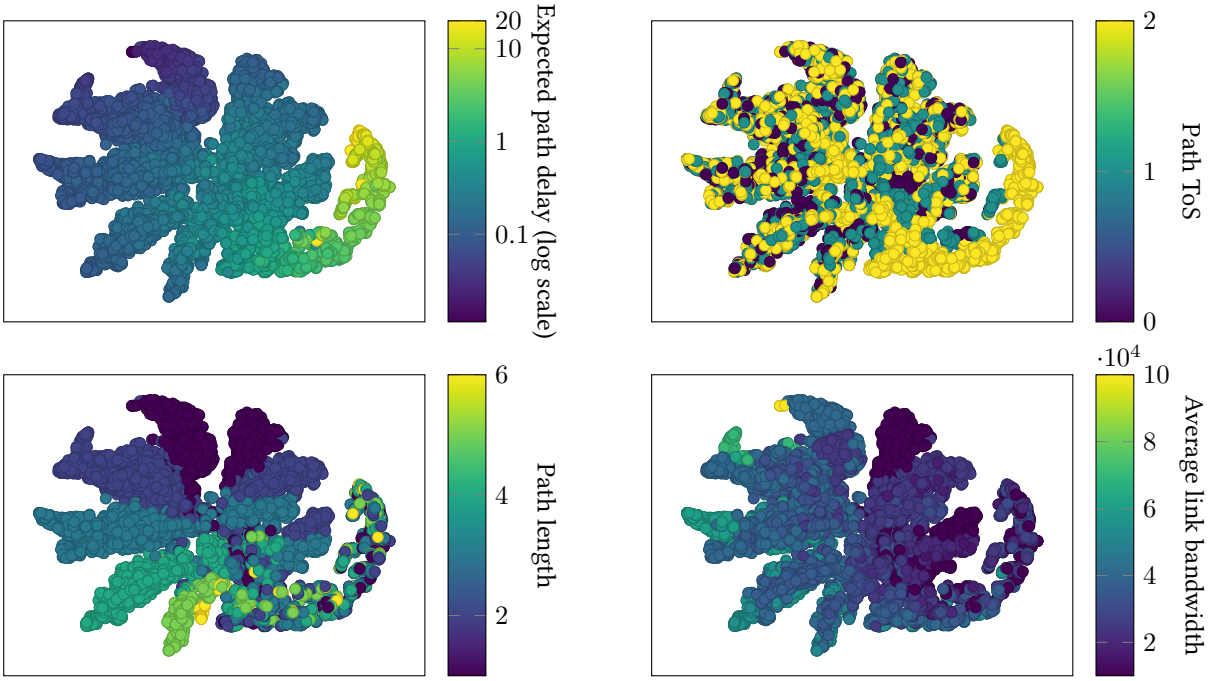


Fig. 3. 2-components t-SNE visualization of paths' hidden states, colored by expected path delay (top left), path type of service (top right), path length in number of links (bottom left) and average link bandwidth within each path (bottom right). We can clearly see that these characteristics are correctly embedded in paths' hidden states.

observed that the predicted delay was biased towards low values when using original delays. Applying the logarithm *before* computing the loss (right plot) reduced the prediction error, especially for high latencies (the dots are closer to the identity function).

Finally, we depict the evolution of the learning rate and the MAPE during the training process in Figure 5. We found that Pytorch's *Cyclic Learning Rate Scheduler* allowed our models to reach lower error values (compared to monotonically-decreasing schedulers). Intuitively, this scheduler avoids local minima by *"jumping"* to unexplored regions in the parameters space using high klearning-rate. This is clearly visible around batch 100 000: after a slow decrease, the learning-rate jumps back to around 5×10^{-4}. The training loss then increases for about 50 000 batches before reaching lower values. We note that the loss over the validation dataset follows the loss over the training dataset with a reasonable difference, showing no sign of overtraining.

5 CONCLUSION

In this document we described our solution to the Graph Neural Networking Challenge 2020 organized by the Barcelona Neural Networking Center. We proposed a new way to transform a network topology with the associated routing information to a set of bipartite graphs that can be used for Graph Message-Passing Convolutions. Based on the proposed bipartite graphs, we constructed specific state aggregations to fully exploit the available features and compute relevant hidden states for network's paths, links and nodes. Finally, we provided training parameters that enabled us to win the challenge with the minimal average relative error.

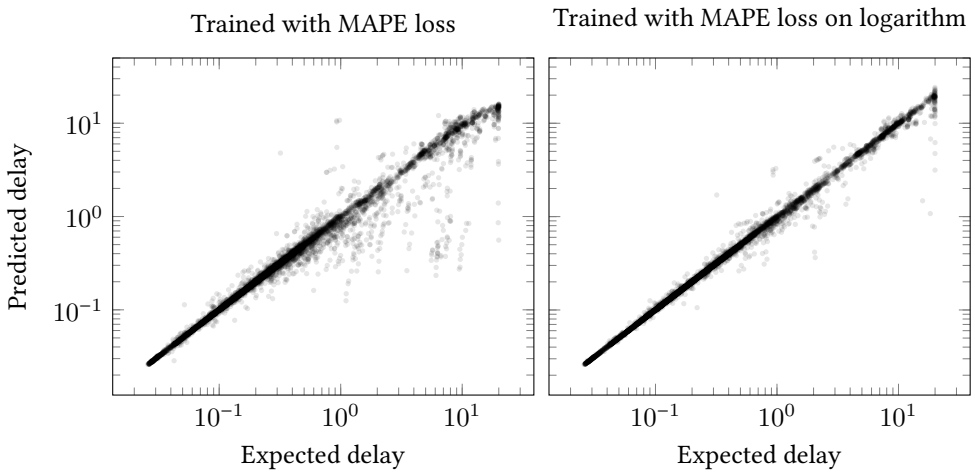


Fig. 4. Predicted vs. Expected delay for some paths in the validation dataset (each dot is a prediction). The left plot shows the result after training on the MAPE loss applied on the original expected delay. The right plot shows the result after training on the MAPE loss applied on *the logarithm of the expected delay*. This second option is less biased towards low predictions.

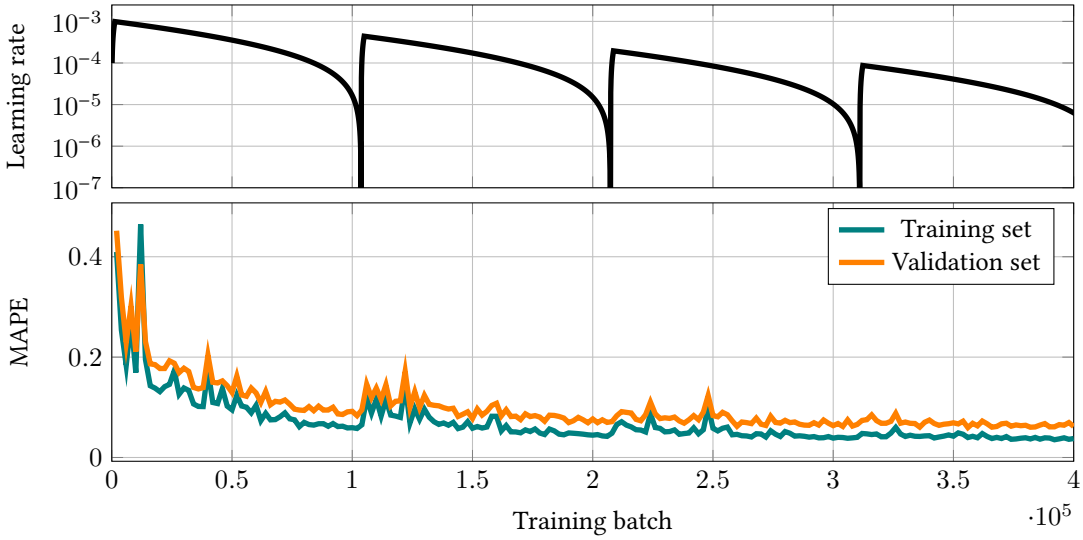


Fig. 5. Learning rate and MAPE loss during the first batches of training. We use Pytorch's Cyclic Learning Rate scheduler.

REFERENCES

[1] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, International Convention Centre, Sydney, Australia, 1263–1272. http://proceedings.mlr.press/v70/gilmer17a.html

[2] Focus Group on Machine Learning for Future Networks including 5G. 2019. *Machine learning in future networks including IMT-2020: use cases.* ITU. Supplement 55 to Y.3170 Series, October 2019.

[3] PyTorch geometric. 2020. *PyTorch geometric - MessagePassing.* https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.message_passing.MessagePassing

[4] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2019. Unveiling the potential of Graph Neural Networks for network modeling and optimization in SDN. In *Proceedings of the 2019 ACM Symposium on SDN Research*. 140–151.