Reproducing: Parameterized Explainer for Graph Neural Network

Anonymous Authors

Reproducibility Summary

Scope of Reproducibility

The main claims that are reproduced in this report are:

1. The PGExplainer is able to correctly identify the ground-truth motif responsible for node and graph classification of a given GNN.
2. The PGExplainer is able to achieve a maximum AUC of 0.987 for node classification and a maximum AUC of 0.926 for graph classification, both with a standard deviation that is a maximum of 0.021.
3. The PGExplainer is able to generate explanations for the given node classification tasks in 24 milliseconds or less, and graph classification tasks in 80 milliseconds or less.

Methodology

The provide codebase, which had a TensorFlow implementation, from the original PGExplainer paper has been used to reproduce their experiments. To replicate their work, the codebase has also been reimplemented to a PyTorch framework. All datasets are tested 10 times to find the average AUC and inference time.

Results

The TensorFlow implementation is able to find and show the correct motifs for all the tested datasets. The PyTorch implementation is able to do the same, except for the MUTAG dataset. The AUC for node classification is higher than stated in the paper for the TensorFlow implementation, the graph classification AUC is mostly similar. The inference time that was found using the PyTorch implementation seems to be in the same ballpark as the results shown in the original study.

What was easy

The paper was well written, which made it easy to understand the concepts and techniques that were used. On top of that, the models were precisely described and in great detail, this made the implementation of the models much easier.

What was difficult

Even though the reimplementation of TensorFlow into PyTorch was not a big obstacle, the rest of the code was not very structured or well written. A number of inconsistencies were found between the code and the paper, mostly in mentioned hyperparameters. Next to that, the provided code did not support GPU processing out of the box.

The last dataset that was used in the original study, the MUTAG dataset, was very big, resulting in some computational problems. Even though the computational problems were managed eventually, the model could not be tested properly on this dataset due to its size.

Submitted to ML Reproducibility Challenge 2020. Do not distribute.

1 Introduction

Many underlying data structures from different areas of science and engineering (e.g. data mining and molecular biology) can be represented in graphs (Scarselli et al. [2008]). These graphs usually have a similar structure in that they all contain nodes and edges. An example of such a graph could be a social network of people that are in contact with each other. In the graph, the persons are denoted by the nodes and their relationships are denoted by the edges. Graphs like these are often researched to find certain patterns or structures within them. A powerful tool used to study graphs are Graph Neural Networks (GNNs), which are a type of Neural Network that directly work on the structure of a graph (Scarselli et al. [2008]). They adapt a message passing design to learn the representation of the nodes by aggregating representation vectors of its neighbors. This design makes the model able to find the features of the nodes and the topology of the graph (Luo et al. [2020]). GNNs are widely applied for graph analysis, because of their high performance and interpretability (Zhou et al. [2018]).

Even though GNNs are useful and efficient, they can be considered a black box, as predictions made by the model are often hard to understand for humans. The model looks at a combination of node features and graph topology to make its predictions, a combination of these 2 factors can thus be seen as an explanation of a prediction. In the work of Ying et al. [2019] a model, the GNNExplainer, has been introduced that is able to find these explanations. This model takes a trained GNN and its predictions as input, and it returns an explanation for the prediction in the form of a subgraph of the original graph along with a subset of the most important node features. However, the problem with this model is that it is instance specific and is thus not able to give a general explanation of how the GNN produces its predictions. It has to be retrained for every new prediction it makes. Consequently, the GNNExplainer has a high computation time making it unfeasible for real life application where multiple graphs need explaining.

A more recent implementation of a model that can explain the predictions of GNNs, is the PGExplainer (Luo et al. [2020]). This model builds on the GNNExplainer by utilizing a generative probabilistic model, which has shown the ability to learn underlying structures from graphs. These underlying structures can in turn be used to explain the predictions that a GNN provides. To be able to explain predictions on a collection of instances, the PGExplainer is parameterized and trained using a deep neural network. Since the neural network parameters are shared across all the explained instances, the PGExplainer is able to provide model-level explanations for the individual instances. This global view of the GNN allows the explainer to explain entire model structures, instead of only single instances. It thus does not need to be retrained when a new graph from the same dataset is presented, but can quickly infer an explanation. (Luo et al. [2020]).

This report contains the reproduction of the experiments done in the paper of Luo et al. [2020]. The authors of the paper provided a codebase that is used to run their experiments. The code is originally implemented in a TensorFlow framework, for this reproduction report it is reimplemented to a PyTorch framework. The provided code and the paper each also present a different set of hyperparameters. The ones present in the code are used for the reproduction, as they obtain better results.

2 Scope of reproducibility

The paper of Luo et al. [2020] introduces a new explainer for GNNs, the PGExplainer, which outperforms a preceding model, the GNNExplainer. The results that were found using the PGExplainer show an improvement in AUC over the GNNExplainer, on explaining both node and graph classifications for the datasets: BA-Shapes, BA-Community, Tree-Cycles, Tree-Grid, BA-2motifs and MUTAG.

Below we specify the central claims made in the PGExplainer study, which we will attempt to reproduce.

1. The PGExplainer is able to correctly identify the ground-truth motif responsible for node and graph classification of a given GNN.
2. The PGExplainer is able to achieve a maximum AUC of 0.987 for node classification and a maximum AUC of 0.926 for graph classification, both with a standard deviation that is a maximum of 0.021.
3. The PGExplainer is able to generate explanations for the given node classification tasks in 24 milliseconds or less, and graph classification tasks in 80 milliseconds or less.

The results will be both reproduced and replicated as per ACM guidelines (ACM [2020]). For the reproduction we will perform the experiments using their publicly available codebase, and done by a different team on different hardware. As part of the replication we rewrite their code from TensorFlow to Pytorch, which allows us to run the experiments without the use of any artifacts provided by the original authors.

3 Methodology

As stated earlier, the authors made their codebase available on a public GitHub repository[1]. Of note is that the provided codebase differed from the architecture and hyperparameters specified in the paper. The experimental results achieved by this code improved on the results reported in the paper, which led us to the decision to use this new code for our reproducibility assessment. The specific changes they made will be expanded upon in the model description section below. The reproduction of the experiments in TensorFlow is straight-forward, as the provided code is run on our hardware without any changes to fairly verify their results.

For the replication experiments the code is reimplemented to a PyTorch framework, using the original code as a template. This is done to test whether the obtained results are independent of the framework that is used in the original paper. In contrast to the framework, the models, datasets and hyperparameters remained the same[2] as given in the codebase.

Both the TensorFlow and the PyTorch implementation of the models are run 10 times per model with a different seed per run to get the average result and the standard deviation, as was the case in the original paper. The models are tested on their Area Under ROC Curve (AUC) score and their inference time. The inference time is the time that is needed for the explainer to generate a motif for a new graph and thus does not include the training time of the model. The AUC score is used in the original paper and earlier literature to report the accuracy of the explanations. We concur with the usage of this metric, as it is especially fit for this task. This is because the generated explanations need enough true positives to yield the correct motif, and few false positives to not include too many uninformative nodes. AUC provides us with a good way to evaluate this trade-off as the generative model outputs probabilities for each edge, for which an artificial threshold has to be set at, for example, 0.5 if we were to use accuracy as a metric. (Huang and Ling [2005]).

3.1 Model descriptions

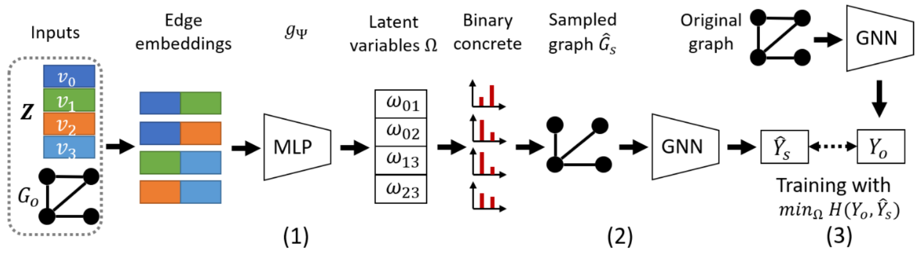


Figure 1: PGExplainer model structure as described below

The experimental setup consists of two models. The first model is a classifier consisting of a Graph Convolutional Network[3] for embedding and a fully connected layer for classification. The classifier differs slightly for the node and graph classification tasks. The second model is a feed-forward neural network, which is the central part of the PGExplainer, responsible for the latent variables used to generate the explanatory subgraphs. The mutual information between this subgraph and the input graph is used as a loss to train the explainer network. A schematic representation made by the original authors is given in figure 1.

The structure of the GNN layer is notated with GNN(a, b, c) with a as input dimension, b as output dimension and c as activation function. The structure of a fully connected layer is denoted with FC(a, b, c), with a, b and c having the same meaning as for the GNN layer. In the code the original authors also use two BatchNormalization layers (BatchNorm) and skip connections in the form of concatenations (Concat), which are not specified in the paper. The structure of the node classification model is GNN(10, 20, ReLU)-BatchNorm-Concat-GNN(20, 20, ReLU)-BatchNorm-Concat-GNN(20, 20, ReLU)-Concat-FC(60, #labels, softmax). A change was made here in the PyTorch implementation, as the obtained AUC increased significantly when the tensors were concatenated after every GNN layer instead of twice after the BatchNorm layer and once after the final GNN layer. Trying this on the TensorFlow implementation did not result in major improvements, as the AUC was already highly optimized. This point will be further expounded upon in the discussion.

[1]https://github.com/flyingdoog/PGExplainer

[2]Except for the learning rate on some experiments, as PyTorch learned much faster than TensorFlow

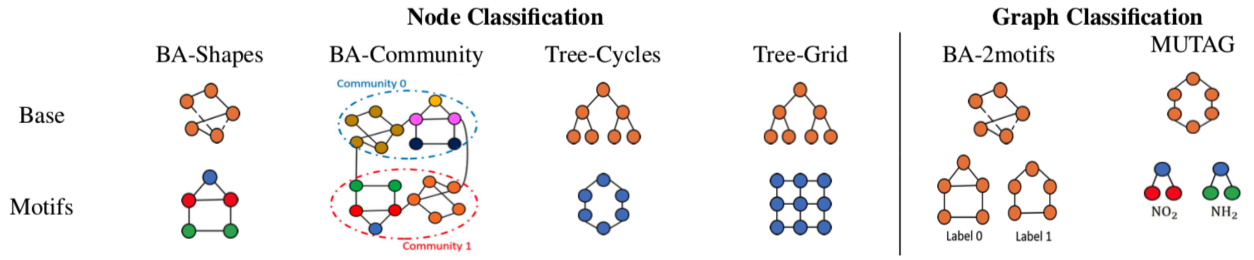
[3]Specified simply as a GNN in the original paper

For graph classification the BatchNorm and Concat layers are not used, but an additional maxpooling layer is specified, and additionally a sum aggregation layer is added in the code. These are added because the input tensors for graph classification contain an additional channel dimension, which has a size of 25 for the BA-2motifs dataset. The maxpooling and summation are both performed on the feature dimension of size 20 of the output of the previous layers, resulting in the BA-2motifs case in two tensors with shape batch size $\times 25$. These two layers are concatenated resulting in a tensor of shape batch size $\times 50$, which is then fed into the final prediction layer. The original paper incorrectly specifies the size of the input dimension of the final layer to be 40, we presume because this does not need to be explicitly specified in TensorFlow, while it does in Pytorch. The final structure of the graph classification model in the code is GNN(10, 20, ReLU)-GNN(20, 20, ReLU)-GNN(20, 20, ReLU)-Maxpooling Summation-FC(50,#labels, softmax).

The structure of the explainer is FC(#input, 64, ReLU)-FC(64[4], 1, Linear). Befor the difference between node and graph classification is that the node classification model has as #input 60 and graph classification has #input 50.

3.2 Datasets

Figure 2: **Base graph and motif structures**



This figure shows the base graph structures and all the corresponding motifs for all the different tested datasets.

The datasets that were used by in the preliminary paper can be categorized into two categories, node classification datasets and graph classification datasets. The node classification datasets that were used, are the same as the ones that were used in the paper in which the GNNExplainer was proposed. For node classification the goal is to classify the nodes, whose values are dependent on their relation to a motif. A motif is a characteristic part of the graph that is used to make predictions, which can thus act as an explanation for the classification. The 4 node classification datasets are all synthetically made, the structure of the graphs and motifs can be seen in figure 2. (1) The BA-shapes dataset has a total of 300 nodes in the form of a Barabasi-Albert (BA) graph with 80 added house shaped motifs. The motifs are connected to randomly selected nodes in the graph, after which additional random edges are added. Nodes in this graph can have 4 different labels, base nodes get the label 0 and the top, middle and bottom of the house motif get the label 1, 2 and 3, respectively. (2) The BA-community graph is a combination of 2 BA-shapes graphs. Features are assigned to every node via two Gaussian distributions. The nodes are classified with 1 of the 8 classes depending on the community membership they are in and the structural role they play. (3) The Tree-Cycles dataset contains a binary tree of a total of 8 levels. A set of 80 motifs in the form of a 6-noded cycle are added to random nodes in the graph. (4) The Tree-Grid has the same base graph as Tree-Cycle, but 80 3-by-3 grid motifs are randomly added instead of 80 6-noded cycles 80.

For graph classification a total of 2 different datasets were used in the preliminary paper. The goal of graph classification is to correctly classify a graph in one of the given types. This is done by finding the motif in the graph that characterizes a specific class of graph. The structure of the graphs and motifs can again be seen in figure 2. (1) The BA-2motif dataset is, like the node classification graphs, syntactically generated. It contains 1000 graphs with a BA base structure. Half of the total amount of graphs are attached with a house motif and the other half with a 5-node cycle motif. The graphs are classified with 1 of the 2 classes, specified by the added motif. This dataset was not used in the paper in which the GNNExplainer was proposed. (2) The MUTAG dataset is the only real world dataset that is used in the original paper. This dataset contains 4337 different molecule graphs, each of which is labeled with 1 of the 2 classes: mutagenic or nonmutagenic. Those graphs that contain a carbon ring with the chemical groups NH_2 or NO_2 are classified as mutagenic, while the nonmutagenic graph have no defining motif, but can also have carbon rings. The two chemical groups can therefore be seen as the motif in the graphs, while the carbon ring does not convey any information.

All datasets are divided into a training, validation and test set. The training set contains 80% of the data, the validation set contains 10% and the test set contains 10%. Table 1 shows the specifications of the different datasets.

[4]The original paper defines this incorrectly as 20

Table 1: **Graph details**

	BA-Shapes	BA-community	Tree-Cycles	Tree-Grid	BA-2motifs	MUTAG
Number of graphs	1	1	1	1	1000	4337
Avg. nodes	700	1400	871	1231	25	30.32
Avg. edges	4110	8920	1950	3410	51.4	61.54
Number of labels	4	8	2	2	2	2

This table shows an overview of all the graphs that are used in this report. The left 4 columns show the node classification graphs and the right 3 columns show the graph classification graphs.

3.3 Hyperparameters

The GNN classifier uses the Adam optimizer, with an initial learning rate of 1.0×10^{-3} for the node GNN and 1.0×10^{-2} for the graph GNN. All weight variables are initialized with Xavier initialization (Glorot and Bengio [2010]). The node GNN is trained for 1000 epochs and the graph GNN for 5000 epochs.

The PGExplainer also uses the Adam optimizer, but with an initial learning rate of 3.0×10^{-3} for node classification and 5.0×10^{-2} for graph classification. These learning rates were found to be too high on some datasets for the PyTorch implementation and were thus set to 1.0×10^{-4} for Tree-Grid and to 1.5×10^{-4} for BA2-motifs. This will be further explained in the discussion. The coefficient of size regularization is set to 0.05 and entropy regularization is set to 1.0. The model is trained for 30 epochs for all datasets. The temperature that is used follows the annealing schedule $\tau^{(t)} = \tau_0(\tau_T/\tau_0)^t$, with $\tau_0 = 5.0$ and $\tau_T = 2.0$.

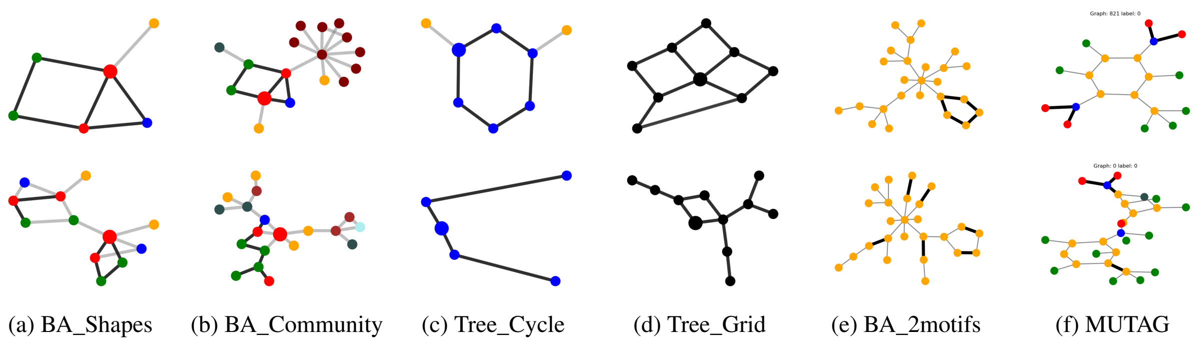
3.4 Experimental setup

To run the multitude of experiments, the supercomputer Lisa from the provider SURFsara has been used. This supercomputer has been made available for students of the University of Amsterdam (UvA) and Vrije Universiteit Amsterdam (VU) to be able to run experiments for projects. The GPU that is used is a quad Scalable Link Interface (SLI) GeForce 1080Ti, each with 11 Gbps of memory speed. The code used is publicly available on the github: https://github.com/afalbrecht/FACTAI21/tree/master/PGExplainer

4 Results

4.1 Qualitative results

Figure 3: **TensorFlow motifs**

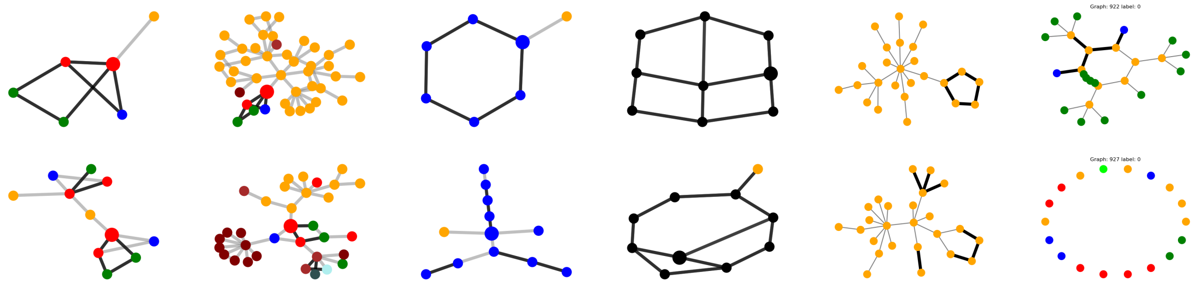


(a) BA_Shapes (b) BA_Community (c) Tree_Cycle (d) Tree_Grid (e) BA_2motifs (f) MUTAG

This figure shows correct (top) and incorrect (bottom) motifs found with the TensorFlow implementation of the code for every dataset.

Figure 3 shows the motifs that were found using the TensorFlow implementation of the code. As can be seen, the model is able to find the correct motifs for the different datasets. The claim that explanations mapping to the ground truth motifs could be found is verified and reproducible. However, upon further inspection, there are also instances in which the PGExplainer fails to find the ground-truth motif. A manual qualitative evaluation was conducted and the following the following qualitative results were found for each dataset: (1) BA-Shapes performed the best, with only up to 5/50 uninformative graphs over the different iterations. (2) BA2-community found the ground truth relatively consistently, but as it was a noisier dataset, it included the uninformative BA part of the graph more often, namely 15 times out of 50. (3) Tree-Circles performed less consistent, with only 20/50 graphs qualifying as circles and the rest consisting of lines or single nodes. It does consistently find the informative (blue) nodes, which explains why the AUC and loss are still optimal. This disconnect between the metrics and provided explanation shows a crucial weakness inherent in the proposed method. (4) Tree-grid only got 10/50 strictly correct, as most others consisted of only three squares. (5) BA-2motifs was very unstable at generating explanations, as can also be seen in the quantitative results. This resulted in some seeds yielding only 2/50 mistakes, while other seeds would yield only 5/50 correct explanations. (6) This instability was less prevalent in the MUTAG dataset, as most explanations were correct. This is positive, as it shows the usefulness of the explainer on a real world dataset.

Figure 4: **PyTorch motifs**



(a) BA_Shapes (b) BA_Community (c) Tree_Cycle (d) Tree_Grid (e) BA_2motifs (f) MUTAG

This figure shows correct (top) and incorrect (bottom) motifs found with the PyTorch implementation of the code for every dataset. The only exception for the top motif being is the shown motif for the MUTAG dataset.

Figure 4 shows the motifs that were found using the reimplemented PyTorch model. The results seem comparable to the results shown in figure 3. The PyTorch implementation is able to find the correct motifs on all the datasets, except for the MUTAG dataset. Not all generated images were correct, with high failure rates for BA-shapes, Tree-Grid and BA-2motifs, while the other two datasets showed similar failure rates as the TensorFlow implementation. The reasons for these errors will be discussed in the discussion. Still, for all datasets besides MUTAG, the correct motifs are found by the PyTorch implementation, making the original paper replicable for those datasets.

4.2 Quantitative results

4.2.1 Average explanation AUC

Table 2: **Average AUC**

	BA-Shapes	BA-Community	Tree-Cycle	Tree-Grid	BA-2motifs	MUTAG
TensorFlow	$0.999 \pm 8.251 \times 10^{-5}$	0.994 ± 0.0005	0.997 ± 0.0003	0.992 ± 0.0002	0.827 ± 0.1341	0.899 ± 0.1651
Pytorch	$0.999 \pm 5.691 \times 10^{-5}$	0.996 ± 0.0004	0.933 ± 0.0839	0.708 ± 0.1127	0.516 ± 0.1625	0.385 ± 0.0020
Improve	0%	0.4%	-6.42%	-28.62%	-37.61%	-57.17%

This table shows the average AUC for the different graphs for the two neural network frameworks. The AUC is calculated by running every model 10 times, each with a different random seed. The accuracy improvement from TensorFlow to PyTorch is shown in percentage.

Table 2 shows the results of the AUC test using both the TensorFlow and PyTorch implementations for all the different datasets. For the node classification tasks, the TensorFlow implementation shows to be extremely accurate, in fact it shows to be more accurate than the scores in the original paper. This discrepancy can be partly explained by the updated code given by the authors, which improves upon the scores reported in the paper on every dataset. But interestingly enough for Tree-Grid and BA-Community our results are better (respectively 0.06 an 0.04 better on AUC) than when using their pre-trained weights, which might be explained by our different hardware and multiple seeds, though it remains uncertain. For the graph classification tasks, the TensorFlow implementation shows to be slightly worse in the BA-2motifs dataset and slightly better in the MUTAG dataset. The decrease in the BA-2motifs set concurs with the instability touched upon above, as its AUC fluctuated between 0.5 and 0.98 on different seeds. This will be returned to in the discussion.

The PyTorch implementation performs well on the BA-shapes, BA-community and the Tree-cycle dataset for the node classification graphs. However, the AUC score for the Tree-grid dataset is much lower than the score of the TensorFlow implementation, it has a decrease of 28.62%. It also shows to have a high standard deviation, the AUC score for the Tree-grid dataset can fluctuate between values of 0.941 and 0.58. This is due to an instability in the loss function used in the paper, which will be further discussed in the discussion.

The PyTorch implementation performs worse than the TensorFlow implementation when it comes to the graph classification tasks. For the BA-2motifs dataset there is a decrease in AUC score of 37.61%. Only one seed tested seed was able to score an AUC of 0.95, the seed 1234 which is interestingly also used by the original authors. This does again attest to the explainer on the BA-2motif dataset, which was not touched upon in the original paper. Thus we were not able to fully replicate or reproduce their results in the instance of the BA2-motif dataset. MUTAG also wasn't replicable, but the other node classifier dataset were replicable with some minor adjustments, as will be touched upon in the discussion.

4.2.2 Average explanation time

Table 3: **Average inference time (ms)**

	BA-Shapes	BA-Community	Tree-Cycle	Tree-Grid	BA-2motifs	MUTAG
Pytorch	3.611 ± 0.58	3.675 ± 0.6	1.752 ± 0.43	1.861 ± 0.42	2.179 ± 0.569	74.711 ± 10.251

This table shows the inference time in milliseconds for the different graphs for the PyTorch framework. The inference time is calculated by running every model leach with a different random seed.

Table 3 shows the inference time results for the different datasets for the PyTorch implementation. Since both the original paper and code did not provide an explicit way to time the inference operation, the TensorFlow implementation is omitted. However, the found inference times seem to be in the same ballpark as the inference times reported in the preliminary study.

5 Discussion

In conclusion: reproducibility claim 1 was able to be reproduced for all datasets and replicated for all except MUTAG. Reproducibility claim 2 was able to be reproduced for all datasets except for BA-2motif, which had a higher standard deviation than was reported in the original paper. Not all datasets were able to be replicated, but qualifications for this statement will be made below.
Reproducibility claim 3 was able to be replicated, but not able to be reproduced due to an oversight in our implementation as discussed below.

Several points remain to be discussed:

1. BA2-motifs instability the only major point that hampers the reproducibility, and it is suspected by us that this has to do with the dataset itself, as it did get the right subgraphs, but did not correctly rank the most important edges. This could be due to latent mutual information in these edges, that also allow the classifier to classify, besides the motifs which stand as ground truths. 2. Pytorch seemed to learn much faster in the cases of BA2-motifs and Tree-Grid, which resulted in loss-hacking behaviour. This can be seen in the case of Tree-Grid, as here it finds the optimum after 3 epochs, then plateaus for a while, and then starts dropping heavily. This problematic behaviour was caused by the original implementation of the size_loss, which put a penalty on the size of the subgraphs by summing the edgeweights in the adjacency matrix. To get this loss down the model then started decreasing these continuous variables by an order of 10 every epoch, resulting in pathological AUC behaviour with decent graphs. Changing this loss to merely adding up the amount of non-zero edge weights immediately improved stability, but it was also found that decreasing the learning rate by an order of 10 helped circumvent the issue as well. This was not reported on our paper, but can be seen by decreasing the learning rate. 3. BA-shapes got worse pictures than expected, which was a problem encountered the day before the paper had to be finalized, as earlier they were replicated. This turned out to be due to the concatination of the batchnorm as touched upon above, which when done in the pytorch friendly way results in better pictures. This is also not shown in the report due to time issues. 4. The tensorflow inference time was not reported due to an oversight in measuring the duration, which was fixed in pytorch but not able to be fixed in tensorflow before the deadline.

The overall result of this reproduction study mainly supports the claims that are made in the preliminary study. However, there still is room for improvement for finding evidence to support the claims that were made.

To further strengthen the claims made in the paper, ideally the models should be tested on a new dataset on top of the datasets that are used in the original study. This tests the generlizability of the model, which validates the overall usefulness of such a model. Next to that, it would be interesting to test this model on a larger dataset and on larger graphs.

Since the authors provided the code that was used for their study, the reimplementation of the model into the PyTorch framework was mostly based on that provided code. However, in hindsight, the provided code did not suit as a good template for the PyTorch implementation. For further inspection of the model, it would be interesting to completely reimplement the model from scratch, so that the model is computationally stronger. Testing this could provide more information about the power of the model.

Due to a misconception, the code that was provided by the authors was not run on a GPU, but instead on a CPU. This was done because the code that was provided, did not support GPU out of the box. So, to fully test their claims about computation speeds, the provided code should be ran on a GPU.

5.1 What was easy

The paper from Luo et al. [2020] was mostly well written and detailed. The appropriate use of images and tables is well done and are structurally good. The formulas for the PGExplainer are also accurately described, which makes interpreting the model easier.

5.2 What was difficult

Unfortunately, there were several inconsistencies in the code and in the paper that had to be solved. To begin with, adjusting the provided code from a TensorFlow framework to a PyTorch framework was harder than initially expected. Some functions that are available in the TensorFlow framework are non-existent or slightly different in the PyTorch framework, making the task time consuming. This was combined with the fact that certain pieces of code are poorly written. To give an example, the temperature hyperparameters described in section 3.3 did not update for the Tree-Cycle graphs as it should have done, and stayed at 5 for the entire training sequence.

Another difficulty was that some aspects that were described in the paper did not match how the code worked. This are some of the inaccuracies that were found. (1) The maxpooling layer that is supposedly used in the graph classification model was not present in the code. The reason for this is unknown. (2) For the node classification model, the code had a batchnorm layer that was not described in the paper. This layer has been kept for the rewriting of the code. (3) The paper described the structure of the explainer model to be FC(#input, 64, ReLU)-FC(20, 1, Linear). This is ofcourse impossible, because the output of the first layer needs to match the input of the second layer. The code showed that the input of the second layer had to be 64. (4) The amount of epochs for the training of the different explainer models is described to be 30. However, the amount of epochs for the training of the BA-shapes dataset has been set to 10 in the code. The reason for this is again not known.

5.3 Communication with original authors

During this study there has been no contact with the authors of the preliminary paper, since there were no big issues with the code or the paper.

References

ACM. Artifact review and badging - current, 2020. URL `https://www.acm.org/publications/policies/artifact-review-and-badging-current`.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

Jin Huang and Charles X Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering*, 17(3):299–310, 2005.

Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. *Advances in Neural Information Processing Systems*, 33, 2020.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in neural information processing systems*, pages 9244–9255, 2019.

Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.