Geometric Data Analysis : Benchmarking GNNs

Raphaël Bernas
ENSTA Paris - MVA
Palaiseau, France
raphael.bernas@ensta-paris.fr

Maxime Corlay
ENSTA Paris - MVA
Palaiseau, France
maxime.corlay@ensta-paris.fr

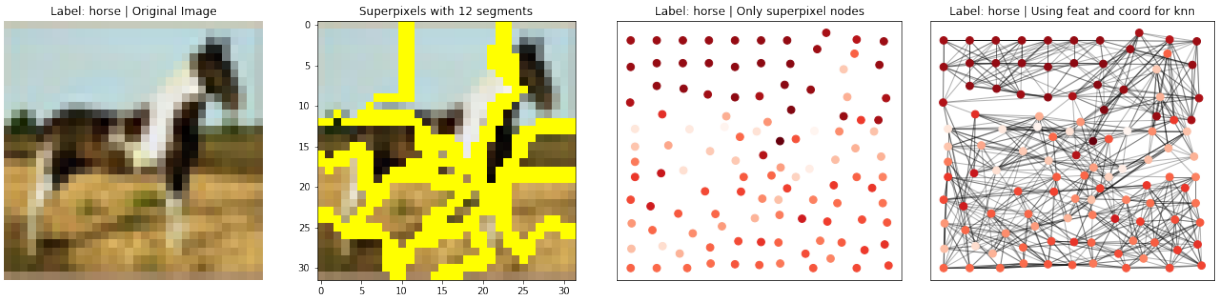

Figure 1: Preprocessing steps to apply a Graph Neural Network (GNN) to an image from the CIFAR-10 dataset. The original image (left) is segmented into superpixels (second : here only 12 have been computed to make the image readable), which are then represented as nodes (third). Finally, a graph is constructed by connecting nodes using k-nearest neighbors based on feature similarity and spatial coordinates (right).

Abstract

Graph Neural Networks (GNNs) are promising in many applied fields. In "Benchmarking Graph Neural Networks" [2], Vijay P. Dwivedi, Chaitanya K. Joshi, et al. propose an overview of some GNN architectures. In [2], they provide benchmarking datasets to fairly compare different models. Our goal is to assess some results written in their paper. As they insist on the importance of the reproducibility of their results, we compare carefully our results with theirs before proposing some personal observations.

Keywords

Graph, Neural Network, Dataset, Benchmark

ACM Reference Format:
Raphaël Bernas and Maxime Corlay. 2024. Geometric Data Analysis : Benchmarking GNNs. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Overview of [2]

To begin this section, let us comment the work presented in the paper [2]. The code associated with this paper defines 8 GNN models which achieved state-of-the-art results at the time the paper was published. Furthermore, the authors did not limit their work to a small number of datasets but included a wide variety. They even anticipated the situation where a user wishes to add his own dataset. Such an extensive overview would obviously require access to multiple GPUs. Thus it is important to thank them for their investment to produce a general and usable paper for the industry. In addition, we commend the choice made to detail a mathematical formulation for each GNN and give to the reader a global view on state-of-the-art methods.

However, the code is presented as perfectly open and machine-agnostic. We believe that this is not the case. They provide two files .yml to load all packages, but when the environment does not already contain the packages, the process is killed due to the excessive number of packages. As a result, we downloaded every package one at a time and it was probably much more time consuming than using directly the .yml file. Moreover, we believe that the code is not accessible to people with few knowledge of computer science. Indeed, most scripts require advanced understanding of Python and Bash. Unfortunately, if one wants to play with parameters, they will need to dive into the code.

Despite these drawbacks, it is appreciated to have precised the environment and all the versions of the libraries because the software stack breaks very often.

1.2 Introduction to Graph Neural Networks (GNNs)

Graph Neural Networks (Overview on GNNS [1] : page 9-20) operate on graphs, represented as $G = (V, E, X)$, where:

- V is the set of nodes, with $|V| = N$,
- $E \subseteq V \times V$ is the set of edges,
- $X \in \mathbb{R}^{N \times d}$ is the node feature matrix, where $\mathbf{x}_i \in \mathbb{R}^d$ represents the d-dimensional features of node i.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The graph structure is captured by the adjacency matrix $A \in \mathbb{R}^{N \times N}$, where $A_{ij} = 1$ if an edge exists between nodes i and j, and $A_{ij} = 0$ otherwise [6]. If the edges are weighted, A_{ij} is the weight of edge $i \to j$.

1.2.1 Learning on Graphs. The goal of GNNs is to learn node embeddings $H \in \mathbb{R}^{N \times p}$ or to classify nodes, edges or even the graph itself. This is achieved by iteratively updating node representations [4]. At layer $t+1$, the update for node i is:

$$\mathbf{h}_i^{(t+1)} = f\left(\mathbf{h}_i^{(t)}, \mathrm{AGG}\left(\{\mathbf{h}_j^{(t)} : j \in \mathcal{NB}(i)\}\right)\right),$$

where:

- $\mathbf{h}_i^{(t)}$ is the node i representation at layer t.
- $\mathcal{NB}(i)$ is the set of neighbors of node i.
- $\mathrm{AGG}(\cdot)$ aggregates, combines information from neighboring nodes.
- $f(\cdot)$ is a learnable update function.

1.2.2 Message Passing Framework. For message-passing neural network (MPNN) -a peculiar class of GNNs- updates involve explicit message and update functions [3]. For an edge (i, j), the message $\mathbf{m}_{ij}$ is computed as:

$$\mathbf{m}_{ij} = M\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}, \mathbf{e}_{ij}\right),$$

and the node embedding is updated as:

$$\mathbf{h}_i^{(t+1)} = U\left(\mathbf{h}_i^{(t)}, \sum_{j \in \mathcal{N}(i)} \mathbf{m}_{ij}\right),$$

where $\mathbf{e}_{ij}$ represents edge features, if available.
In [2] : GIN, GCN, GraphSAGE, and GAT are MPNN based model.

1.2.3 Explicit Form : Matrix Formulation. An efficient representation of GNNs with matrices, as in Graph Convolutional Networks (GCNs) [6], updates node embeddings at layer $t+1$ as:

$$H^{(t+1)} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(t)}W^{(t)}\right),$$

where:

- $\tilde{A} = A + I$ is the adjacency matrix (with self-loops).
- $\tilde{D}$ is the degree matrix of $\tilde{A}$.
- $W^{(t)}$ is the weight parameter matrix.
- $\sigma(\cdot)$ is an activation function.

1.2.4 Supervised Learning Objective. For node classification tasks, the final embeddings $\mathbf{h}_i^{(T)}$ are used to predict labels $\mathbf{y}_i$ via a softmax classifier. The cross-entropy loss is minimized:

$$\mathcal{L} = -\sum_{i \in V_{\text{train}}} \sum_{c=1}^{C} y_{ic} \log \hat{y}_{ic},$$

where V_{train} is the set of training nodes, y_{ic} is the true label, and $\hat{y}_{ic}$ is the predicted probability.

This framework enables GNNs to generalize neural networks to non-Euclidean data.

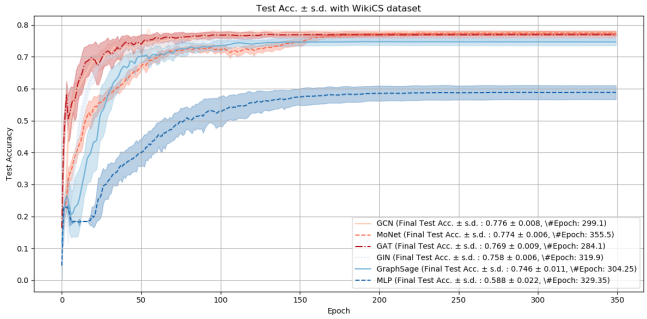

Figure 2: Test Acc. ± s.d. with WikiCS

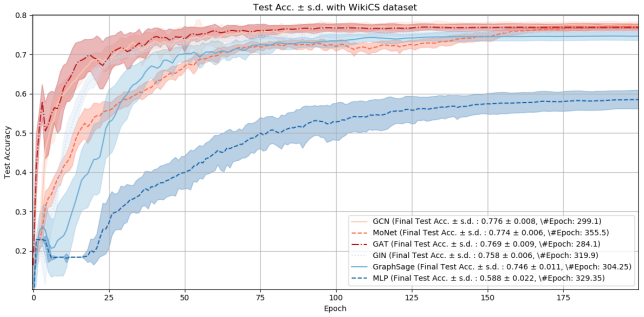

Figure 3: Test Acc. ± s.d. with WikiCS (zoom)

1.3 Benchmarking overview

We detail here the process we are going to follow in this work.

First, we are going to detail how we have realised a test of reproducibility for the GNNs benchmarking. Such a reproductability test has been conducted on the WikiCS dataset and SBM.

Then we are going to dive further into the code and conduct some experiments on the ZINC dataset to highlight the limits of the process made by Vijay P. Dwivedi, Chaitanya K. Joshi, et al.

Finally we added to the benchmarking a new model : Graphormer.

2 WikiCS

2.1 The dataset

This dataset consists of a unique graph extracted from Wikipedia, where **nodes** represent webpages and **edges** represent hyperlinks between them.

The graph is a simplified web, classified into **10 main classes** related to Computer Science. Each webpage belongs to one of the following topic classes:

(1) COMPUTATIONAL LINGUISTICS
(2) DATABASES
(3) OPERATING SYSTEMS and OPERATING SYSTEMS TECHNOLOGY
(4) COMPUTER ARCHITECTURE
(5) COMPUTER SECURITY, COMPUTER NETWORK SECURITY, ACCESS CONTROL, DATA SECURITY, COMPUTATIONAL TRUST and COMPUTER SECURITY EXPLOITS
(6) INTERNET PROTOCOLS

Model	our #Epochs	#Epochs of [2]
GCN	299.1	299.85
MoNet	355.5	355.81
GAT	284.1	275.48
GIN	319.9	321.25
GraphSage	304.25	303.68
MLP	329.35	322.46
GatedGCN	OOM	OOM

Table 1: Reproducibility of mean #Epochs by split

Model	our mean Test Acc.	mean Test Acc. of [2]
GCN	0.776	0.775±0.009
MoNet	0.774	0.774±0.007
GAT	0.769	0.769±0.008
GIN	0.758	0.759±0.006
GraphSage	0.746	0.748±0.010
MLP	0.588	0.595±0.023
GatedGCN	OOM	OOM

Table 2: Reproducibility of mean final Test. Acc.

(7) COMPUTER FILE SYSTEMS
(8) DISTRIBUTED COMPUTING ARCHITECTURE
(9) WEB TECHNOLOGY, WEB SOFTWARE and WEB SERVICES
(10) PROGRAMMING LANGUAGE TOPICS, PROGRAMMING LANGUAGE THEORY, PROGRAMMING LANGUAGE CONCEPTS and PROGRAMMING LANGUAGE CLASSIFICATION

The goal of this task is to do **node classification**, identifying the class that best corresponds to each node. Each node feature is a **300-d embedding** of the text associated with the webpage.

For further details, please refer to Péter Mernyei and Cătălina Cangea's paper [7].

2.2 Comparison of models : the method

- **Test Set:** 50% of the nodes are used for testing. Once this test set is defined, it remains strictly separate from the remaining 50% of nodes, which are used for training and validation. Importantly, test nodes are **never** included in any splitting process.
- **Training and Validation Sets:** The remaining 50% of nodes are allocated for training and validation. To ensure robustness, we conduct statistical analysis over different training sets. Therefore, we divide this 50% into 5% for training and 45% for validation, and perform 20 different splits. We follow the procedure outlined by Vijay P. Dwivedi, Chaitanya K. Joshi et al. described in [2].
- **Hardware:** All experiments are conducted on a CPU (11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, 2.42 GHz, with 16 GB of RAM). We evaluate seven models: GCN, MoNet, GAT, GIN, GraphSage, MLP, and GatedGCN.

2.3 Comparison of models : reproducibility of [2]

The GatedGCN model returns an out-of-memory (OOM) error, as reported in the work of Vijay P. Dwivedi, Chaitanya K. Joshi, et al. [2]. Table 1 shows that the mean number of epochs is very similar to what was obtained in [2]. Table 2 shows that the mean final Test Accuracy is very similar to what was obtained in [2] : in fact, for each model, our result falls in the interval [acc. ± σ]. This demonstrates a very good level of reproducibility.

2.4 Comparison of models : additional comments compared to [2]

The goal of this section is to go beyond what Vijay P. Dwivedi, Chaitanya K. Joshi, et al. made in [2]. Instead of simply showing the final accuracy results, in Figures 2 and 3 we plot the evolution of the test accuracy for the 6 models. We make two observations: a) It appears useless to go beyond 200 epochs, as the accuracy stabilizes after around this approximate number of epochs. b) The dynamics are interesting: MoNet, which ultimately performs among the best models, is relatively slow in the first epochs, whereas GCN shows a much faster rate of improvement.

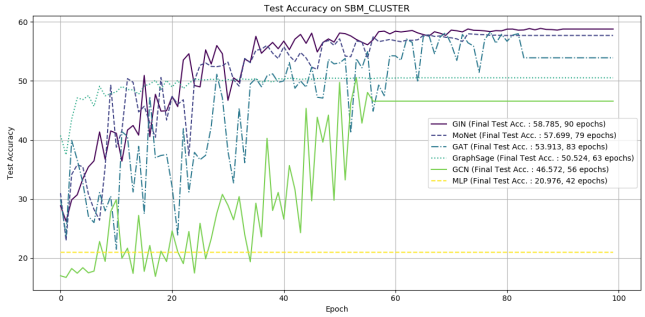

Figure 4: Test Acc. ± s.d. with SBM

3 SBM

3.1 SBM

SBM (Stochastic Block Model) is a method for randomly generating graphs with clusters. The aim is to evaluate the ability of GNN models to detect clusters in a graph, with applications in the social sciences.

3.2 Comparison of models : the method

- **Test Set:** 8.33% of the graphs are used for testing. Once this test set is defined, it remains strictly separate from the remaining nodes, which are used for training and validation.
- **Training and Validation Sets:** Another 8.33% is used for validation, and 83.33% is used for training. The sets are separate and clearly defined (no system of split) for fair comparison between models, as said in [2].
- **Hardware:** All experiments are conducted on a CPU (11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, 2.42 GHz, with 16 GB of RAM). We evaluate seven models: GCN, MoNet, GAT, GIN, GraphSage and MLP.

Model	our #Epochs	#Epochs of [2]
GCN	56	63.50
MoNet	79	76.25
GAT	83	77.75
GIN	90	74.75
GraphSage	63	64.00
MLP	42	42.25

Table 3: Reproducibility of #Epochs

Model	our Test Acc.	mean Test Acc. of [2]
GCN	46.572	47.828±1.510
MoNet	57.699	58.064±0.131
GAT	53.913	57.732±0.323
GIN	58.785	58.384±0.236
GraphSage	50.524	50.454±0.145
MLP	20.976	20.973±0.004

Table 4: Reproducibility of final Test. Acc. by split

3.3 Comparison of models : reproducibility of [2]

Table 3 shows that the number of epochs is near what was obtained in [2], but not perfectly equal. This is probably due to the fact that we did only one run for each model. If we did several runs, we would statistically obtain better results in terms of reproducibility. Table 4 shows that the final Test Accuracy is similar to what was obtained in [2] : in fact, for each model, our result falls in the interval $[\text{acc.} \pm \sigma]$ except for MoNet and GAT (where the gap isn't big). This demonstrates a very good level of reproducibility.

3.4 Comparison of models : additional comments compared to [2]

The goal of this section is to go beyond what Vijay P. Dwivedi, Chaitanya K. Joshi, et al. made in [2]. Instead of simply showing the final accuracy results, in Figure 4 we plot the evolution of the test accuracy for the 6 models. We make two observations: a) It appears useless to go beyond 100 epochs, as the accuracy stabilizes after around this approximate number of epochs. b) The dynamics are interesting: GraphSage rises quickly but then stabilizes and stops increasing.

4 ZINC

4.1 The dataset

The **ZINC dataset** is a graph-based dataset used in machine learning, particularly for GNNs. It is derived from the ZINC database, a collection of molecular compounds designed for virtual screening and drug discovery [5].

In this dataset, molecules are represented as graphs, where:

- **Nodes** represent atoms (e.g., carbon, oxygen, nitrogen).
- **Edges** represent chemical bonds (e.g., single, double, aromatic).
- **Node features** encode atomic properties (e.g., type, charge).
- **Edge features** capture bond characteristics (e.g., bond type, aromaticity).

In the case of GNNs, the task associated with the ZINC dataset is *molecular property regression*. For example, predicting the *penalized logP*, which measures molecular solubility and drug-likeness [5].

4.2 Learning rate study : the method

- **Learning Rate Schedule:** For each gradient-based optimizer, we need to set a learning rate and its schedule. The schedule is made of two informations : an update function which changes the learning rate value and a criteria to change the learning rate value.
- **Peculiar Case of [2]:** In the code introduced by Vijay P. Dwivedi, Chaitanya K. Joshi, et al., they have made a fixed choice that is to use *ReduceLROnPlateau*, a method reducing the learning rate when a monitored metric (here validation loss) stops improving. Unfortunately such a pre-implemented method has its own limitations and the code does not allow us to change that.
- **Study Process:** After training different models on ZINC and monitoring the learning rate, we produced a graph : Refer to Figure 5. Then, we proceed to wonder if the already set hyperparameters for this *ReduceLROnPlateau*-method are correct to produce a *fair* benchmark.
- **Hardware:** For those experimentations, as we will possibly have longer training process, we will use a GPU (Nvidia RTX 3050 Laptop). We evaluate seven models : MLP, GraphSage, GCN, MoNet, GAT, GatedGCN and GIN.

4.3 Learning rate study : The results

In the *ReduceLROnPlateau*-method, the learning rate is updated after a number of epochs. Thus, in such a setting the learning rate is a step function of the epochs. Even if this appears to be simple, this method involves multiple hyperparameters which have not been taken into account in [2].

The most important one in our current study is the *learning rate schedule patience* (T_{lr}). This parameter sets for how long we stay on a "plateau" of our step function if the monitored metrics does not evolve. In all the experiments conducted by Vijay P. Dwivedi, Chaitanya K. Joshi, et al. on ZINC, we have that $T_{lr} = 5$. This is not an anecdotic choice : some models -particularly in the case of graph learning- need a longer time to adapt and properly understand the underlying structure of the graph-data. Those models could produce better results if allowed more time.

Thus, we decided to reproduce their experiments with a new *learning rate schedule patience* value $T_{lr} = 25$. Then, we trained the different models on ZINC.

The results :

- Learning rate evolution graphic : Refer to Figure 5.
- Mean Absolute Error for each models : Refer to Table 5.

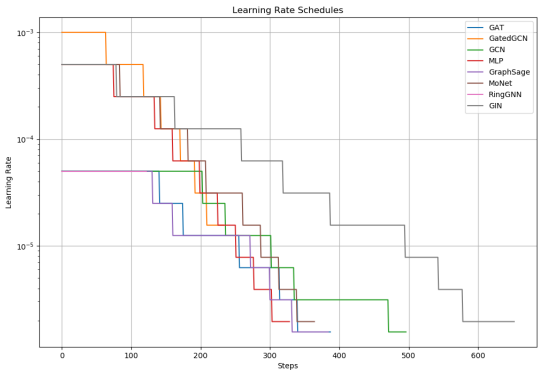

Figure 5: Evolution of learning rate with *ReduceLROnPlateau*-method for multiple models trained on ZINC (y-axis is log-scaled).

Model ($L = 4$)	#Param	ZINC Test MAE	ZINC Train MAE	#Epoch (Time)
MLP	106,970	0.670	0.591	328 (1.62s/e)
GraphSage	105,031	0.455	0.270	384 (3.06s/e)
GCN	103,077	0.361	0.267	496 (29.9s/e)
MoNet	106,002	0.358	0.209	364 (4.25s/e)
GAT	102,961	0.499	0.338	387 (4.41s/e)
GatedGCN	105,735	0.428	0.262	233 (3.58s/e)
GIN	103,079	0.314	0.198	652 (5.56s/e)
Graphormer[†1]	328,481	0.546	0.438	392 (7.87s/e)
†1 : Refer to Section4.5.				
GIN ($L = 16$)[†2]	509,549	0.263	0.195	564 (13.7s/e)
†2 : Refer to Section4.4.				

Table 5: Benchmarking results for ZINC for graph regression. Results (MAE: Mean Absolute Error) are obtained over 1 run with *seed*0 of Dwivedi, Koshi et al.'s experiment. Blue signifies that the value is lower than the value obtained in [2]. Red signifies that the value is higher. For MAE, the lower the value, the better the performance.

4.4 Learning rate study : Why should they have dived deeper ?

The first obvious observation we can do is that with a higher patience, models take more epochs to converge. This is expected because we permit our models to stay longer on each "plateau". Then using Table 5 and Figure 5 together, we observe an interesting phenomenon : GIN appears to takes its time on each "plateau" and if given much more time to converge, GIN attains greater results (it is even the best model out of all the one tested).
If we compare with the results obtained in [2], GIN is the one that improves the most. This is something that possibly could have been guessed from the results. Indeed, Vijay P. Dwivedi, Chaitanya K. Joshi, et al. also test GIN with 16 layers and observe that such a model is worse than GIN with 4 layers (There GIN $L = 16$ attained a test MAE of 0.526). A potential reason of that is exactly because GIN takes much more time to correctly adapt to data and increasing the numbers of parameters also increase the times required to do so.
Obviously, such an hypothesis could be false but we believe that such an observation could appear alarming specifically because we are conducting a benchmarking.
Finally, we are going to train GIN with $L = 16$ layers for $T_{lr} = 25$. As we do not dispose of the powerfull GPU used in [2], we did not compute models for $L = 16$ but in the peculiar case of GIN, it appears now to be relevant to do so. After training we obtain :

$$\text{GIN}(L = 16) \quad \text{Test } MAE = 0.263 \quad \text{Train } MAE = 0.195 \quad e = 564$$

Those results are far better than the one obtained by GIN ($L = 16$) in [2]. They make GIN ($L = 16$) one of the best model trained over ZINC in the settings : epochs capped at 1000 and training time at 12h. We want to highlight here that only one hyperparameter preset has been changed. With the only change, the table is completely changed. And it is because this hyperparameter is so important that the results have been altered drastically.

4.5 New model on ZINC : Graphormer

4.5.1 Graphormer: Graphormer is a model first introduced in June 2021, after the publication of Vijay P. Dwivedi, Chaitanya K. Joshi, et al. The model is introduced by a Microsoft team in [8]. Such a model used -in a new way- transformers which were reputed to be less efficient on graphs and achieved great results.

4.5.2 Performance: Graphormer has shown great performance on graph dataset. It even won the 1st place for quantum prediction track of Open Graph Benchmark Large-Scale Challenge (KDD CUP 2021). Thus, we computed a lighter and adapted version of it for our ZINC dataset graph regression task. Figure 6 shows the architecture of the Graphormer model.

In the context of this peculiar benchmarking, it appears that Graphormer is not a "good model", See Table 5. But as detailed above, Graphormer obtains impressive results in other settings: more layers, more parameters, more time to train (Increased T_{lr}), etc.

4.6 On the importance of settings

In the code, there is no possibility of changing the max number of epochs nor the time limit. The peculiar task of comparing models leads them to make a general parameters configuration process which cannot completely take into account the specificity of each model. They made the effort of giving the T_{lr} but did not explicitely explain there choice nor did they realise test on this peculiar but important hyperparameter.
The goal of this section was not to undermine the work realised by Vijay P. Dwivedi, Chaitanya K. Joshi, et al. but to remark that such a benchmarking process is limited and should not be taken as a fundamental truth. Unfortunately, as Yoshua Bengio is involved in this paper, some could believe such works are complete overview of GNNs. As we have shown so far, it is not the case.

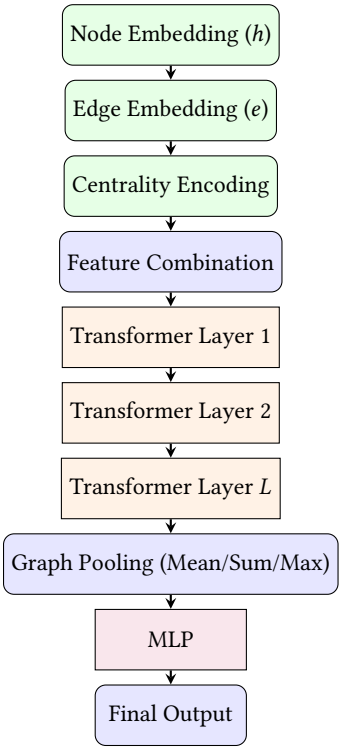

Figure 6: Graphormer model architecture.

5 Conclusion

So far, we have seen that even if the Benchmarking-GNNs code is not "machine agnostic" as it was promised, it is fairly easy to reproduce the exact same experiments they conducted. This shows a great effort in making reproducible work and it is something to praise.

What is mostly striking is the fact that the obtained results are reliable. Indeed, the estimated average error on the accuracy is coherent with the one obtained by Vijay P. Dwivedi, Chaitanya K. Joshi et al.

Unfortunately, we have made many observations here while plotting different graphic to get a better view of the training evolution that were not looked upon by the original author. And some are not anecdotic because they question the overall process followed during the paper.

There is obviously an infinite number of ways to improve a benchmarking. Adding new model would be an interesting direction. Adding new datasets too as the code makes it possible. But we believe that a benchmarking oriented on the training method and with moving hyperparameters would be already a good and generalizable work.

References

[1] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (Jun 2018). arXiv:1806.01261 [cs.LG] https://arxiv.org/abs/1806.01261 Comprehensive review of graph networks and relational learning.

[2] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2022. Benchmarking Graph Neural Networks. *arXiv preprint arXiv:2003.00982v5* (Dec 2022). arXiv:2003.00982v5 [cs.LG] https://arxiv.org/abs/2003.00982v5 Accessed: 2024-12-02.

[3] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *Proceedings of the 34th International Conference on Machine Learning (ICML)* (Jul 2017). arXiv:1704.01212 [cs.LG] https://proceedings.mlr.press/v70/gilmer17a.html Published in ICML 2017.

[4] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. *Advances in Neural Information Processing Systems (NeurIPS)* (Dec 2017). arXiv:1706.02216 [cs.SI] https://arxiv.org/abs/1706.02216 Presented at NeurIPS 2017.

[5] John J. Irwin and Brian K. Shoichet. 2012. ZINC: A Free Tool to Discover Chemistry for Biology. *Journal of Chemical Information and Modeling* 52, 7 (2012), 1757–1768.

[6] Thomas Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (Feb 2017). arXiv:1609.02907 [cs.LG] https://arxiv.org/abs/1609.02907 Published at ICLR 2017.

[7] Péter Mernyei and Cătălina Cangea. 2020. Wiki-CS: A Wikipedia-based benchmark for Graph Neural Networks. *arXiv preprint arXiv:2007.02901v2* (Jul 2020). arXiv:2007.02901v2 [cs.LG] https://arxiv.org/abs/2007.02901v2 Comments: Graph Representation Learning and Beyond workshop (ICML 2020); corrected incorrect metrics due to issue in experiment implementation.

[8] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do Transformers Really Perform Badly for Graph Representation?. In *Thirty-Fifth Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=OeWooOxFwDa

6 Appendix A : Setup and Installation Instructions

As we explained before, the instructions given in the repository cannot be exactly followed in order to succeed in running the code. Thus, to reproduce the experiments or run the benchmarks, follow these installation and setup steps. The following instructions assume you have conda installed.

6.1 Environment Setup

(1) Create a new conda environment (only required the first time):

```
conda create --name benchmark_gnn
```

(2) Activate the environment:

```
conda activate benchmark_gnn
```

(3) Add necessary channels to your conda configuration (only required the first time):

```
conda config --add channels pytorch
conda config --add channels dglteam
conda config --add channels conda-forge
conda config --add channels anaconda
conda config --add channels defaults
```

You can verify that the `pytorch` channel was successfully added by running:

```
conda list pytorch
```

6.2 Install Dependencies

Install the following dependencies (for both CPU and GPU setups): We recommand doing it step-by-step for the process not to be killed.

You can add $-y$ after *conda install* to make it answer "yes" to each process when installing these packages.

```
conda install python=3.7.4 python-dateutil=2.8.0 pip=19.2.3
conda install pytorch=1.6.0 torchvision=0.7.0 pillow=6.1
conda install dgl=0.6.1 numpy=1.19.2 matplotlib=3.1.0
conda install tensorboard=1.14.0
conda install tensorboardx=1.8 future=0.18.2
conda install absl-py networkx=2.3
conda install scikit-learn=0.21.2 scipy=1.3.0
conda install notebook=6.0.0 h5py=2.9.0 mkl=2019.4
conda install ipykernel=5.1.2
conda install ipython=7.7.0 ipython_genutils=0.2.0
conda install ipywidgets=7.5.1 jupyter=1.0.0
conda install jupyter_client=5.3.1
conda install jupyter_console=6.0.0 jupyter_core=4.5.0
conda install plotly=4.1.1 scikit-image=0.15.0
conda install requests=2.22.0
conda install tqdm=4.43.0 cudatoolkit=10.2 cudnn=7.6.5
conda install dgl-cuda10.2=0.6.1
```

Note: This installation may take a significant amount of time.

To verify the installation of specific packages, use commands like:

```
conda list torchvision
```

For `tensorflow` and related packages, use `pip` (ensure to use `==` for version specification):

```
pip install tensorflow==2.1.0 tensorflow-estimator==2.1.0
pip install tensorboard==2.1.1 ogb==1.2.2
```

You can verify the installation of `tensorflow` with:

```
pip show tensorflow
```

6.3 Download Dataset

Clone the benchmark repository and download the dataset:

```
git clone
https://github.com/graphdeeplearning/benchmarking-gnns.git
cd data/
bash script_download_molecules.sh
cd ..
```

6.4 Run Benchmarks

To run the experiments, use the following commands:

CPU Execution:

```
python main_molecules_graph_regression.py \
    --dataset ZINC \
    --config
  'configs/molecules_graph_regression_GatedGCN_ZINC_100k.json'
```

GPU Execution:

```
python main_molecules_graph_regression.py \
    --dataset ZINC --gpu_id 0 \
    --config
  'configs/molecules_graph_regression_GatedGCN_ZINC_100k.json'
```

Note: GPU execution may require adapting the environment for remote GPUs (e.g., on Colab or similar platforms).