The Journal of Open Source Software

PyDGN: a Python Library for Flexible and Reproducible Research on Deep Learning for Graphs

Federico Errica [1¶], **Davide Bacciu** [2], **and Alessio Micheli** [2]

1 NEC Laboratories Europe, Germany **2** University of Pisa, Italy ¶ Corresponding author

DOI: 10.21105/joss.05713

Software

- Review
- Repository
- Archive

Editor: Arfon Smith

Reviewers:

- @idoby
- @sepandhaghighi

Submitted: 26 June 2023
Published: 01 October 2023

Summary

The use of standardized evaluation procedures is a key component in the Machine Learning (ML) field to determine whether new approaches grant real advantages over others. This is especially true for fast-growing research areas, where a substantial amount of literature relentlessly appears every day. In the graph machine learning field, some evaluation issues have already been brought to light and partially addressed, but a general-purpose library for rigorous evaluations and reproducible experiments is lacking. We therefore introduce a new Python library, called `PyDGN`, to provide users with a system that lets them focus on models' development while ensuring empirical rigor and reproducibility of their results.

Statement of need

To date, the graph ML community (Bronstein et al., 2017; Hamilton et al., 2017; Micheli, 2009; Scarselli et al., 2009; Sperduti & Starita, 1997; Wu et al., 2020) has already developed benchmarking software to re-evaluate existing models on a fixed set of datasets (Errica et al., 2020; Hu et al., 2020; Liu et al., 2021; Shchur et al., 2018). In addition, existing libraries such as PyTorch Geometric (PyG) (Fey & Lenssen, 2019), Deep Graph Library (DGL) (Wang et al., 2019), and Spektral (Grattarola & Alippi, 2021) provide the building blocks of Deep Graph Networks (DGNs) (Bacciu et al., 2020), also known as message-passing architectures (Gilmer et al., 2017), effectively acting as the backbone of most graph ML software packages. Other libraries, such as GraphGym (You et al., 2020), allow the user to explore the design of existing DGNs by running hyper-parameter tuning experiments in parallel, but the customization is mostly limited to the models and does not allow, for instance, a modular extension of data splitting techniques, evaluation strategies, or experiments with a custom logic. This limits the ability of a researcher to use these libraries to carry out new and possibly original research.

The community therefore lacks a software library that is specifically dedicated to ensuring reproducibility and replicability of experiments *without* compromising the flexibility required by our everyday research. To fill this gap, we have developed `PyDGN`, a Python library for Deep Graph Networks research. `PyDGN` builds upon `PyTorch` (Paszke et al., 2019) and `PyTorch Geometric` (PyG) (Fey & Lenssen, 2019) to handle graph-structured data and reuse efficient implementations of machine learning models. It exploits Ray to run experiments in parallel (also on clusters of machines) and it supports GPU computation for faster executions. Our goal is to help practitioners and researchers to focus on the development of their models and to effortlessly evaluate them under fair, robust, and reproducible experimental conditions, thus mitigating empirical malpractices that often occur in the ML community (Errica et al., 2020; Lipton & Steinhardt, 2018; Shchur et al., 2018). `PyDGN` has already been used in a number of research projects that have been published at top-tier venues, as listed in the official GitHub repository.

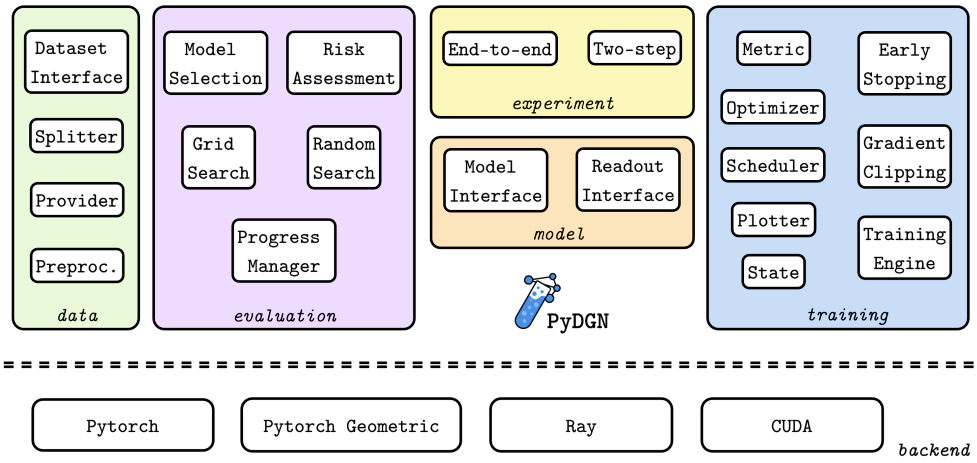

Figure 1: PyDGN is logically organized into different modules that cover specific aspects of the entire evaluation's pipeline, from data creation to a model's risk assessment.

We refer the reader to Figure 1 for a visual depiction of the main components. We remark that all modules, with the exception of the one responsible for evaluation (due to its standard behavior), are readily extensible and promote rapid prototyping through code reuse.

How to use it

Users can easily prepare and launch their evaluations through *configuration files*, one for the data preparation and the second for the actual experiment. In the former, the user specifies: **i)** how to split the data; **ii)** the dataset to use; and **iii)** optional data transformations. In the second file, the user indicates: **i)** data and splits; **ii)** hardware devices and parallelism; **iii)** hyper-parameter configurations for model selection; **iv)** training-specific details like metrics and optimizer. Dedicated scripts prepare the data, its splits, launch the experiments and compute results. The PyDGN documentation helps the user understand the main mechanisms through tutorials and examples.

Data preparation

A recurrent issue in the evaluation of ML models is that they are compared using different data splits. The first step to reproducibility is thus the creation and retention of such splits: we provide code that partitions the data depending on the required graph/node/link prediction scenarios. Since splitting depends on the type of evaluation procedure, we cover *hold-out*, *k-fold*, and *nested/double k-fold* cross validation, which are the most common evaluation schemas in the ML literature.

To create and use a dataset, we provide an interface to easily specify pre-processing as well as runtime processing of graphs; we extend the available dataset classes in PyG to achieve this goal. In addition, a data provider automatically retrieves the correct data subset (training/validation/test) during an experiment, making sure the user does not involuntarily leak test data into training/validation. An example on how to split a dataset to carry out a 10-fold cross validation with inner hold-out model selection is shown below:

```
splitter:
  root: DATA_SPLITS
  class_name: pydgn.data.splitter.Splitter
  args:
    n_outer_folds: 10
```

```
n_inner_folds: 1
seed: 42
stratify: True
shuffle: True
inner_val_ratio: 0.1
outer_val_ratio: 0.1
test_ratio: 0.1
```

Evaluation procedures

PyDGN is equipped with routines that remove the burden of performing model selection and risk assessment from the users. This reduces chances of empirical flaws and favors reproducible experiments. After model selection, the best configuration (with respect to the validation set) is re-trained and evaluated on the test set. In addition, a start-and-stop mechanism can resume execution of unfinished experiments when the whole evaluation is interrupted. These procedures are completely transparent to the user and handled in accordance to the data splits.

Experiment templates

We define an abstract interface for each experiment that consists of two methods, *run_valid* and *run_test*. The first is called during the model selection, and the second is called during risk assessment of the model. This should act as a reminder that the user cannot access the test data when performing a model-selection procedure (that is, *run_valid*), thus reducing the chances of test data leakage. Our library ships with two standard experiments, an end-to-end training on a single task and two-step training where first we compute unsupervised node/graph embeddings and then apply a supervised predictor on top of them to solve a downstream task. These two implementations cover most use cases and ensure that the different data splits are used in the correct way.

Implementing models

To implement a DGN in our library, it is sufficient to adhere to a very simple interface that specifies initialization arguments and the type of output for the prediction step. The user wraps the interface around a PyG model to have it immediately working. This strategy allows the user to focus entirely on the development of the model regardless of all the code necessary to run the training pipeline, akin to what happens in (You et al., 2020). The library will automatically provide the current hyper-parameter configuration to be evaluated to the model in the form of a dictionary `config`. To create a new model, the user simply has to subclass `ModuleInterface` and implement the two methods

```
class MyModel(pydgn.model.interface.ModelInterface):
    def __init__(self,
                 dim_node_features, dim_edge_features, dim_target,
                 readout_class, config: dict):
        ...

    def forward(self, data):
        ...
```

where `data` is a PyG `Batch` object containing a batch of input graphs.

Training via publish-subscribe

The interaction between all components of a training pipeline is perhaps the trickiest part to implement in any machine learning project. One false step, and nothing works. PyDGN's training engine implements all boilerplate code regarding the training loop, and it relies on the publish-subscribe design pattern to trigger the execution of callbacks at specific points in the

training procedure. Every metric, early stopping, scheduler, gradient clipper, optimizer, data fetcher, and stats plotter implements some of these callbacks; when a callback is triggered, a shared state object is passed as argument to allow communication between the different components of the training process. For instance, extending the MeanSquaredError loss to compute its logarithm is as easy as doing

```
class LogMSE(pydgn.training.callback.metric.MeanSquareError):

    @property
    def name(self):
        return "LOG Mean Square Error"

    def compute_metric(self, targets, predictions):
        mse_metric = super().compute_metric(targets, predictions)
        return torch.log(mse_metric)
```

The new metric can be already referenced and used in the configuration file by referrint it as `my_metric_file.LogMeanSquareError`. It will also be automatically logged on Tensorboard by the `Plotter` callback if the latter is enabled.

Key Design Considerations

Two important aspects make PyDGN a user-friendly choice for everyday research.

First, PyDGN's use of configuration files allows users to add new hyper-parameters without needing to modify any extra code. This feature simplifies the customization of a model while prototyping it to keep the user focused on the main task, that is, the definition and implementation of a model, ignoring all the boilerplate code used to forward the new hyper-parameters to the model's class initializer.

Additionally, the publish-subcribe design pattern keeps the codebase organized and flexible. Users can connect different components as needed using a shared state object, making it adaptable to various use cases. Most of the time, refining a training loop only requires subclassing the desired callbacks, simplifying the overall development process.

In summary, PyDGN's configuration files and event-based design pattern enhance its usability, making it a practical tool for researchers by simplifying customization and maintaining code flexibility. These features empower researchers to efficiently achieve their research goals.

Acknowledgements

We acknowledge contributions from Antonio Carta, Danilo Numeroso, Daniele Castellana, Alessio Gravina, Francesco Landolfi, and support from Marco Podda during the genesis of this project. The authors would like to thank the reviewers Ido Ben-Yair and Sepand Haghighi for the constructive feedback that greatly improved the paper and the installation procedure.

References

Bacciu, D., Errica, F., Micheli, A., & Podda, M. (2020). A gentle introduction to deep learning for graphs. *Neural Networks*, *129*, 203–221. https://doi.org/10.1016/j.neunet.2020.06.006

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric deep learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, *34*(4), 25. 18–42. https://doi.org/10.1109/MSP.2017.2693418

Errica, F., Podda, M., Bacciu, D., & Micheli, A. (2020). A fair comparison of graph neural networks for graph classification. *8th International Conference on Learning Representations*

(ICLR).

Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. *Representation Learning on Graphs and Manifolds Workshop, International Conference on Learning Representations (ICLR)*.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 1263–1272.

Grattarola, D., & Alippi, C. (2021). Graph neural networks in TensorFlow and keras with spektral. *IEEE Computational Intelligence Magazine*, *16*(1), 99–106. https://doi.org/10.1109/MCI.2020.3039072

Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin*, *40*(3), 52–74.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., & Leskovec, J. (2020). Open graph benchmark: Datasets for machine learning on graphs. *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)*, 22118–22133.

Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *arXiv Preprint arXiv:1807.03341*.

Liu, M., Luo, Y., Wang, L., Xie, Y., Yuan, H., Gui, S., Yu, H., Xu, Z., Zhang, J., Liu, Y., Yan, K., Liu, H., Fu, C., Oztekin, B. M., Zhang, X., & Ji, S. (2021). DIG: A turnkey library for diving into graph deep learning research. *Journal of Machine Learning Research*, *22*(240), 1–9.

Micheli, A. (2009). Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, *20*(3), 498–511. https://doi.org/10.1109/TNN.2008.2010350

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., & others. (2019). Pytorch: An imperative style, high-performance deep learning library. *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, *20*(1), 61–80. https://doi.org/10.1109/TNN.2008.2005605

Shchur, O., Mumme, M., Bojchevski, A., & Günnemann, S. (2018). Pitfalls of graph neural network evaluation. *Workshop on Relational Representation Learning, Neural Information Processing Systems (NeurIPS)*.

Sperduti, A., & Starita, A. (1997). Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, *8*(3), 714–735. https://doi.org/10.1109/72.572108

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., & Zhang, Z. (2019). Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv Preprint arXiv:1909.01315*.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*. https://doi.org/10.1109/TNNLS.2020.2978386

You, J., Ying, Z., & Leskovec, J. (2020). Design space for graph neural networks. *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)*.