University Ulm | Faculty Computer Science | Institute of Databases and Information Systems

Online Learning of Graph Neural Networks: When Can Data Be Permanently Deleted?

Technical Report

Benedikt Franke,
Tobias Zielke

Ulm, December 2, 2020
Supervisor Ansgar Scherp, Lukas Galke

Abstract

This document describes the *IncrementalGNNs* framework used during the work on Galke, L.; Franke, B.; Zielke, T.; Scherp, A.: Online Learning of Graph Neural Networks: When Can Data Be Permanently Deleted? 2020. The framework provides the necessary experimental apparatus to compare different graph neural network architectures and sampling methods on temporal graphs. We provide an overview over the features of the framework and give usage instructions. We also provide some insights on how to extend the framework for future users.

Contents

1 Introduction

This document is intended as the technical counterpart to [1]. It describes the usage and implementation of the experimental framework used for the experiments in the paper. The framework makes it possible to compare different graph neural networks and graph sampling techniques on different data sets. We will provide a short overview over the implemented Graph Neural Networks and Graph Sampling Techniques. We also describe the implementation process of our new *ClassBalanced Sampler*. We will give usage information and examples for the framework as well as information on the already included graph data sets. Furthermore, the output format of the framework and some provided utility scripts to process the output files are explained.
Finally, we give some starting points for possible adaptions of the framework.

2 Implemented models and sampling methods

The framework provides a number of GNN-architectures and GNN-sampling methods. For an overview which sampling method can be combined with other network architectures, see Section 3.5. In this report, we only briefly outline the methods, for a more in-depth discussion see [1].

2.1 Network architectures

GraphSage-Mean concatenates a vertex' hidden representation with the mean of the representations of the vertex' neighbours before applying the weight matrix and nonlinearity [2].

Graph Attention Network is a representative of anisotropic GNNs, i. e. edges are weighted instead of being treated equally. The embeddings of the vertices are computed by an multi-head attention mechanism [3].

Multi Layer Perceptron We include a standard MLP architecture for comparison purposes. This MLP does not use information about the graph structure and only predicts labels for each vertex based on its features.

Jumping Knowledge uses residual connections from each layer to the output layer. This enables the network to learn the importance of each neighbourhood-hop separately [4].

Simplified GCN computes the n-hop neighbourhood as the n-th power of the normalized adjacency matrix, skipping non-linearities between layers. This makes the network much easier to compute, resulting in much faster training.

2.2 Sampling methods

Stochastic GCN uses a control-variate based algorithm to sample vertices in a way that minimizes the variance of the resulting estimator. This approach has shown to work with extremely small sampling sizes [5].

GraphSAINT samples subgraph-wise instead of vertex-wise. Therefore, a budget m is defined and in a randomwalk fashion vertices are sampled into a subgraph until the budget is depleted. GraphSAINT also modifies the loss-function to account for the importance of different vertices and edges [6].

Class Balanced Sampler (CBS) As an extension/modification to GraphSAINT, we include a subgraph-sampler that leads to a better representation of minority classes in the training data. This is achieved by sampling via a random walk, where nodes are greedily chosen in a way that tries to maximize the class-distributions entropy [1].

2.3 Implementation details of Class Balanced Sampler

Over the course of our work, we tried different libraries as a backbone for the ClassBalanced-sampler. We also experimented with a probabilistic approach to the sampling algorithm, where we build a probabillity distribution over all available nodes depending on the resulting entropy. However, calculating the distribution was very costly, therefore we stayed with the greedy approach.
We first experimented with a pure `numpy`-based version, which was used to confirm that the sampling algorithm indeed manages to maximize the entropy of the sampled subgraphs. However, this versions was too slow to be used for GNN-training, therefore we turned to `cython`, as its numpy-support made a smooth, incremental transition possible. Sadly because `PyTorch` does not have the same cython-support as numpy, the cython sampler was bottlenecked by the conversion between Torch-tensors and cython-memoryviews.
This led us to try a different approach: Instead of writing an entirely new sampler from the ground up, we implemented a sub-class of `torch_geometric.data.GraphSAINTSampler`, which we were already using for the experiments with GraphSAINT anyway. We also switched from computing the subgraphs completly on-demand to a hybrid approach: The sampled vertices are pre-computed before the training starts, and during training we use the base class `GraphSAINTSampler` to compute all edges between the sampled nodes. This approach keeps the effect on training time low, while not consuming too much memory, as we avoid having to store edges in memory. As a final optimization, we employed parallelism by splitting the pre-processing in two steps: First, the starting nodes for all random walks are computed in parallel by class. After that, the random walks are executed in parallel based on the provided start nodes. The results are stacked to a `torch.Tensor` and stored for retrieval during training. Using these optimizations, we were able to cut the effect on training time of our sampler down to acceptable levels and use it for our experiments.

3 Usage information

3.1 Setup and installation

The framework is available via GitHub [7]. After cloning, the majority of dependencies can be installed by executing:

```
$ pip install -r requirements.txt
```

Unfortunately, *dgl* and *PyTorch-geometric* setup depends on the respective CUDA versions and needs to be done independently after this. For installation instructions, see [8] and [9].

The datasets are available via zenodo [1]. Each dataset should be placed in the `data`-subfolder.

3.2 Arguments

The main entry point of the framework is the file `run_experiment.py`. This script is used to run training and evaluation with a given parameter set on a specified data set. It is controlled by a variety of command line arguments. For more information on the paramter choices, see Section 3.5.

run_experiment arguments

model	choose the underlying model. Available options: • gat • gcn • gcn_cv_sc • gs-mean • jknet • mlp • mostfrequent • node2vec • sgnet
sampling	Choose the sampling method. Be aware that some combinations of model + sampling may not be reasonable Available options: • class_balanced • graphsaint_edge • graphsaint_node • graphsaint_rw
variant	A string passed into the result file right after the model column. Could be used to write a comment.
dataset	Specify the used dataset. Available options: • dblp-easy • dblp-hard • dblp-full • pharmabio
t_start	Indicate the first evaluation time step. Default is 2004 for DBLP-{easy,hard} and 1999 for PharmaBio
n_layers	Number of layers/hops
n_hidden	number of trainable units per layer (network dimension)
lr	learning rate
weight_decay	weight decay rate
dropout	dropout probability
initial_epochs	number of initially (pre)trained epochs
anual_epochs	number of epochs trained per year

1 https://zenodo.org/record/3764770

history	temporal window size (how many years of data to keep in history)
gat_out_heads	How many output heads to use for GATs
rescale_lr	Rescaling factor for learning rate after pretraining
rescale_wd	Rescaling factor for weight decay after pretraining
seed	Seed for training
num_neighbors	How many neighbors for control variate sampling
limit	In debug mode: limit the number of papers to lead
batch_size	Number of seed nodes per batch
test_batch_size	Test batch size
num_workers	How many threads to use for sampling
limited_pretraining	Perform pretraining on the first history window
decay	Parameter for exponential decay loss smoothing
save_intermediate	Save intermediate results per year
save	Save results to the provided path
start	Use trained wights for earlier training (warm) or retain from scratch (cold). Available options: • cold • hybrid • legacy-cold • legacy-warm • warm
mc_pool_size	*Not implemented*
mc_pool_smoothing	*Not implemented*
mc_pool_alpha_mc	*Not implemented*
mc_pool_alpha_o	*Not implemented*
walk_length	Walk length for GraphSAINT random walk sampler
saint_coverage	number of nodes to compute GraphSAINT normalization
cbs_njobs	number of jobs used for precomputing CBS-sampling
include_frontier	if set, include all direct neighbors of CBS subgraph

3.3 Examples

Simple MLP on DBLP-easy

```
$ python run_experiment.py --model mlp --dataset dblp_easy
```

100 epochs of GraphSAGE on pharmabio

```
$ python run_experiment.py --model gs-mean --dataset pharmabio --annual_epochs 100
```

Sampling: Use GraphSAINT with JKNet

```
$ python run_experiment.py --model jknet --dataset dblp-hard --sampling graphsaint_rw
```

Simplified GCN with learning rate 0.01 and warm restart

```
$ python run_experiment.py --model sgnet --dataset dblp-hard --start warm --lr 0.01
```

3.4 Datasets

With the repository, three datasets are provided. DBLP-easy and DBLP-hard are temporal graphs based on the DBLP citation network [10] and yield information about citation between scientific publications on several conferences, journals, etc. Pharmabio [11] is a temporal graph of co-authorship from bio- and pharma-tech companies.

Table 2: Total number of vertices $|V|$, number of edges $|E|$ excluding self-loops, number of features D and number of classes $|C|$, number of newly appearing classes $|C_{\text{new}}|$ within the evaluation time steps, the 25,50,75-percentiles of the distribution of temporal differences, along with the total number of time steps T for our datasets.

Dataset	$\|V\|$	$\|E\|$	D	$\|C\|$	percentiles	T
Pharmabio	68,068	2,1M	4,829	7	1, 4, 8	21
DBLP-easy	45,407	112,131	2,278	12 (4 new)	1, 3, 6	25
DBLP-hard	198,675	643,734	4,043	73 (23 new)	1, 3, 6	25

DBLP-easy is used to optimize hyperparameters. DBLP-hard and pharmabio are used to get test results on.

For more details look up Appendix A2 in [1].

3.5 Backends: geometric and dgl

The framework can work with two backends: *dgl* [8] and *PyTorch-geometric* [9] [12]. As different models are implemented based on different backends, the backend will be set automatically based on the parameter choices for `model` and `sampling`.

DGL The *dgl*-backend is used for the network architectures GAT, GS-mean, GCN and MLP. It also provides the sampler Stochastic GCN. This is the default backend that is used as long as there is no necessity to switch backends (see paragraph PyTorch-geometric).

PyTorch-geometric The more recent additions to the framework are implemented using PyTorch-geometric. It provides implementations of JKNet and Simplified GCN, as well as the samplers GraphSAINT and ClassBalanced. If one of these samplers is used, GS-mean, GAT, GCN and MLP are also available in this backend.

Incompatibilities As a consequence of the split between backends, certain combinations of samplers and network architectures are not possible at the moment. Most importantly, GraphSAINT and CBS can only be used with network architectures implemented in PyTorch-geometric (TG). However, at the time of writing we provide TG alternatives of most network architectures implemented in DGL, that are automatically switched in when these samplers are selected.
It is not possible to combine Stochastic GCN with any other network architecture, as it is implemented as a model and uses GCN as a fixed architecture type. Logically, it is therefore also impossible to use GraphSAINT and Stochastic GCN at the same time, even though they are set by different parameters.

4 Result files

4.1 Metrics

The framework currently automatically records accuracy and f1-macro of all experiments. While accuracy provides a good overall-measure of the performance of a selected network configuration, we include f1-macro to account for class-imbalance. This is especially noticeable on DBLP-hard, or when using the ClassBalanced sampler. Accuracy is simply defined as the ratio of correct predictions to total number of predictions:

$$\text{accuracy} = \frac{\#\text{ correct predictions}}{\#\text{ total predictions}}$$

F1-macro accounts for the dataset distribution and, for N classes, is defined as:

$$\text{f1}_{\text{macro}} = \frac{1}{N}\sum_{i=0}^{N}\text{f1}_i$$

with f1_i being the binary f1-score for class i:

$$\text{f1}_i = 2 \cdot \frac{\text{precision}_i \cdot \text{recall}_i}{\text{precision}_i + \text{recall}_i}$$

For more details on the metrics, see the `scikit-learn` user guide[2].
It is possible to change metrics or include other metrics, for this see Section 5.

4.2 Format

The following columns are recorded in the resulting csv-files:

Column name	Explanation
dataset	Name of the used data set
seed	initialisation seed for the training run
model	underlying model
variant	string that is passed by the variant parameter call
n_params	number of trainable parameters
n_hidden	number of hidden units
n_layers	number of layers
dropout	rate of dropout
history	temporal window size
sampling	sampling method
batch_size	number of sampled vertices to from a whole mini-batch
saint_coverage	Number of vertices used to compute GraphSAINT normalization
limited_pretraining	restrict pretraining to limited window size
initial_epochs	epochs used for pertaining

2 https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics

initial_lr	initial learning rate
initial_wd	weight decay during pre-training
annual_epochs	number of training epochs per year
annual_lr	learning rate used during training
annual_wd	weight decay on each timestep
start	cold or warm start
decay	weight decay
year	current year in the training data
epoch	epoch of evaluation (could be more than one evaluation per timestep)
f1_macro	achieved f1-macro score
accuracy	achieved accuracy score

4.3 Processing scripts

tabularize.py [3] This script converts the csv result files into latex tables by grouping the dataset by arguments. Data used in the paper [1] is gathered by using this script, but transformed into the format of the paper manually.

Argument `-g` accepts multiple grouping parameters which are the names the several columns in the result file have.

Argument `-latex` indicates that a latex result file is created. Without a save location, the latex code will only be outputted in the console window.

Argument `-save` will create a output file at the given location

Example call:

```
$ python tabularize.py resultsFolder/results.csv -g history start --latex --save result.txt
```

visualize.py [4] This script creates an accuracy over year plot. The most important parameters are shown in Table 4.

name	explanation
data	at least one file that contains the data to plot must be given
ci	if set, plot 95% confidence interval
sd	if set, plot standard deviation. Incompatible to ci
save	path to save plot

Table 4: Arguments of visualize.py

Example call:

```
$ python visualize.py resultsFolder/results.csv --ci --save plots/results.png
```

The script also takes the arguments `style, hue, markers, row, col, vertlines, noshary` and `aspect`, which get passed directly to the equally named arguments of `seaborn.relplot` [5].

3 https://github.com/BeFranke/Incremental-GNNs/blob/pooling/tabularize.py
4 https://github.com/BeFranke/Incremental-GNNs/blob/pooling/visualize.py
5 https://seaborn.pydata.org/generated/seaborn.relplot.html

extract_params.py [6] This is a small utility script to extract the best parameters for each combination of window size and start-type. It takes a path to a result csv-file, as well as the options `-g` and `-f`.
The `g`-option is used to specify the hyperparameter search space. For example, by passing `-g n_hidden annual_lr`, the script will output the best setting for the number of hidden units and learning rate for each combination of start and window size found in the csv file.
With the `f`-option the number of hidden units can be set to a fixed number. An example use case is, given a csv file where multiple hidden sizes were recorded, one would want to find the best parameters for hidden size 64 only. In this case, `-f 64` can be passed to the script.

Example call:

```
$ python extract_params.py resultsFolder/results.csv plots/results.png -g n_hidden annual_lr
```

5 Possible extensions

In the previous sections, we described the usage of the *IncrementalGNNs*-framework. We now give some hints to future users on how to change or add features. In general, after most adaptions it is necessary to change the possible parameter choices in `run_experiment.py` [7]. This is necessary in order to use your adaptions by specifying command line parameters.

Adding datasets Datasets can be added by simply modifying the `DATASET_PATHS`-variable in `run_experiment.py` to include the new dataset and specifying the path to the data.
A dataset should consist of four files:

- `adjlist.txt` A file containing a `networkx`-adjacency list
- `t.npy` A numpy matrix giving the time stamp of appearance for each node in the graph
- `X.npy` A numpy matrix containing the feature matrix of the graph
- `y.npy` A numpy vector containing the labels of the graph

Changing or implementing new metrics Metrics are returned by the `evaluate`-functions. At the moment, two return values are reserved for metrics. Therefore, to change metrics is is necessary to adapt the respective evaluate-method, i. e. the main evaluate in `run_experiment.py` or the sampling-evaluation in `models/graphsaint.py` [8]. The result file format can be adapted to the new metric by changing `RESULT_COLS` in `run_experiment.py`.

Adding new models Models are defined in the sub-package `models` [9]. It is recommended to derive the new models class from `torch.nn.Module`. If this is done, only two more methods need to be implemented to account for our restart-mechanic: `reset_final_parameters()`, which should reset the weights of the last layer, and `final_parameters()`, which should yield (not return!) weights and bias of the final layer. For examples, see any file from `models`.
After implementing the model, the method `build_model` in `run_experiment.py` needs to be adapted in order to actually construct the model for training.

6 https://github.com/BeFranke/Incremental-GNNs/blob/pooling/extract_params.py
7 https://github.com/BeFranke/Incremental-GNNs/blob/pooling/run_experiment.py
8 https://github.com/BeFranke/Incremental-GNNs/blob/pooling/models/graphsaint.py
9 https://github.com/BeFranke/Incremental-GNNs/tree/pooling/models

Adding samplers Adding samplers is perhaps the most complex task, simply because there are multiple ways to include the sampler in the framework. We describe the most simple one, which is to adapt `models/graphsaint.py` to include another sampler choice.
API-wise, the sampler-class should be iterable, yielding one batch per iteration. This is the only hard requirement to the samplers API. After that, the method `train_saint` in `models/graphsaint.py` must be adapted to construct and use the new sampler. The evaluation does not necessarily need to be changed, as we currently do not sample during evaluation.

6 Conclusion

In this report, we gave technical information on the *IncrementalGNNs*-framework. We gave setup instructions and documented the possible arguments as well as the output format of the main entry point, called `run_experiment.py`. We also explained the included utility scripts that help users to further process their result files. At the end, we also gave some first impressions on how to adapt the framework for new users needs, i. e. by adding new models or changing the recorded metrics of the experiments. We hope that this document helps future users with their research.

References

1. GALKE, L. et al.: Online Learning of Graph Neural Networks: When Can Data Be Permanently Deleted? 2020.
2. HAMILTON, W. L. et al.: Inductive Representation Learning on Large Graphs. *CoRR*. 2017, vol. abs/1706.02216. Available from arXiv: 1706.02216.
3. VELIČKOVIĆ, P. et al.: Graph Attention Networks. In: *ICLR*. 2018.
4. XU, K. et al.: Representation Learning on Graphs with Jumping Knowledge Networks. In: *ICML*. 2018.
5. CHEN, J. et al.: Stochastic Training of Graph Convolutional Networks with Variance Reduction. In: *ICML*. 2018.
6. ZENG, H. et al.: GraphSAINT: Graph Sampling Based Inductive Learning Method. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. Available also from: https://openreview.net/forum?id=BJe8pkHFwS.
7. GALKE, L. et al.: *IncrementalGNN GitHub-Repository*. [N.d.]. https://github.com/BeFranke/Incremental-GNNs, last accessed FILL DATE.
8. DGL CONTRIBUTORS: *DGL Documentation*. [N.d.]. https://docs.dgl.ai/, last accessed 2020-10-22.
9. TORCH-GEOMETRIC CONTRIBUTORS: *PyTorch-Geometric Documentation*. [N.d.]. https://github.com/rusty1s/pytorch_geometric, last accessed 2020-10-22.
10. MELNYCHUK, T. et al.: Effects of university-industry collaborations in basic research on different stages of pharmaceutical new product development. In: *Innovation and Product Development Management Conference*. 2019.
11. TANG, J. et al.: ArnetMiner: Extraction and Mining of Academic Social Networks. In: *KDD '08*. ACM, 2008.
12. FEY, M.; LENSSEN, J. E.: Fast Graph Representation Learning with PyTorch Geometric. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.