THE LONDON SCHOOL OF ECONOMICS AND POLITICAL SCIENCE

Department of Statistics 2019

A Comparative Study on Graph Convolutional Networks

Submitted for the Master of Science,

London School of Economics,

University of London

April 2019

Abstract

Since Kipf and Welling (2016) introduced graph convolutional networks in the setting of semi-supervised node classification, several extensions have been proposed. For assessing the performance of these methods, the majority of the literature relies on an experimental setup introduced by Yang et al. (2016) which consists mainly of small, sparse graphs. The purpose of this study is to analyze the performance of different GCN methods based on a comprehensive range of datasets which is complex with respect to size and sparsity. In particular, we focus on three GCN architectures: basic GCNs, FastGCNs and graph attention networks. Micro F1 scores and training time per epoch are reported for three real-world datasets and five simulated graphs. We find that all methods are comparable in prediction performance. While GAT achieves accuracies up to three percent points faster than the basic GCN framework, it is up to 20 times slower. FastGCN is the fastest method for large and dense datasets if training time per batch is considered.

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

CNN	Convolutional neural network.
GAT	Graph attention network.
GCN	Graph convolutional network.
i.i.d.	independent identitcally distributed.

1 Introduction

Graph Convolutional Networks (GCNs) are universal neural network architectures which operate on graph-structured data. In contrast to traditional convolutional networks, they do not only input feature vectors but also relational information, commonly in form of an adjacency matrix. Networks of this kind have already been applied in diverse research areas. Cucurull et al. (2018) employed GCNs for predicting functional regions for locations on a human brain mesh. Liberis et al. (2018) used GCNs to predict the amino acids of an antibody that participates in binding to a target antigen.

In this paper, we study in particular the basic class of GCNs as introduced by Kipf and Welling (2016). Several extensions of this framework have been introduced ever since. For example, Veličković et al. (2018) allow for learnable weights in their class of Graph Attention Networks (GATs) and Chen et al. (2018) propose a speed-accelerated extension based on importance sampling, called FastGCN.

After Kipf and Welling (2016) assessed the performance of their newly proposed class of models based on the experimental setting of Yang et al. (2016), most following papers (including the aforementioned) continued to mimick this setup, which is mainly based on small and sparse citation datasets.

Our goal is to assess the performance of the three convolutional networks mentioned before for semi-supervised node classification in a broader setting also considering large and dense datasets. The main contribution of this paper is the numerical evaluation and comparison of three GCN architectures with respect to accuracy, training time and scalability based on three real-world and five synthetic datasets of different size and sparsity.

Our results show that all algorithms are comparable in test accuracy. While FastGCN is a little faster for large datasets, GCN is faster for smaller datasets. GATs are the slowest models with barely any benefit over the basic framework. Increasing the sam-

ple size and edge density result in a higher need of computational time and memory for all models, especially GAT and GCN. As a result, GAT and GCN cannot be computed for large graphs if a batched approach is not to be chosen.
Regarding the lack of literature summarizing the contributions to the development of graph convolutional networks, this paper begins by a brief summary of the aforementioned model, before we introduce the experimental setup and evaluate our numerical results.

2 Theoretical Concepts

We consider a graph $G = (V,E)$ with nodes $V = \{v_1,...,v_N\}$, edges E and feature matrix $H_0 = (H_0^i)_{i=1}^N \in \mathbb{R}^{N\times F_0}$. If the nodes of G have no features, feature learning methods such as node2vec (Grover and Leskovec, 2016) or DeepWalk (Perozzi et al., 2014) can be applied or $\mathbf{H}^0$ is set equal to the identity matrix (Kipf and Welling, 2016). The structure of the graph can be summarized by its adjacency matrix $A \in \mathbb{R}^{N\times N}$. Our aim is to find a CNN with L layers which takes as inputs A and H^0 and outputs convoluted features H^L. If we assume homophily in a graph (this defines as close nodes with similar features), the convoluted feature vector H_0^i of a node v_i would ideally depend on the features of v_i and its neighbors.
The idea of GCNs can be generalized by the Weisfeiler-Lehman algorithm: For all nodes $v_i \in V$ the node feature H_i^0 is updated by $H_i^L = hash(\sum_{v_j\in\mathcal{N}_{v_i}} H_j^0)$, where $hash(\cdot)$ is an injective hash function and $\mathcal{N}_{v_i}$ denotes the neighborhood of v_i.

2.1 Basic Framework

According to Kipf and Welling (2016), a graph convolutional layer takes as input a feature matrix H^l and a matrix representation of the graph A and outputs a more abstract feature matrix H^{l+1} according to a propagation rule f. The propagation rule f transforms the node-wise features $H_i^l, i = 1,...,N$ first, then weights them by the matrix W^l. The vectors are then recombined at each node according to A such that the output vector at each node should be a combination of features of the immediate neighbourhood. A baseline propagation rule would thus be

$$f(H^l) = \sigma(AH^lW^l)$$

where $\mathbf{W}^l \in \mathbb{R}^{F^l \times F^{l+1}}$ is the weight matrix for layer l and $\sigma : \mathbb{R}^{N\times F^{l+1}} \to \mathbb{R}^{N\times F^{l+1}}$ is a non-linear activation function such as the ReLu function (Jepsen, 2018) or a softmax function if $l = L$. Since the weights are shared across different nodes according to A, this propagation resembles the filtering in convolutional layers which inspired the name of this framework (Kipf and Welling, 2016). As an effect, nodes that share neighbors tend to have similar feature representations.
Kipf and Welling (2016) define their choice of propagation rule

$$f(H^l) = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^lW^l),$$

where $\tilde{A} = A + I_N$ and $\tilde{D} = diag((\sum_{j=1}^{N}\tilde{A}_{i,j})_{i\in\{1,...,N\}})$ is the degree matrix of $\tilde{A}_{i,j}$ by a first-order approximation of localized spectral filters on graphs according to Hammond et al. (2011). By introducing self-loops in the adjacency matrix, the propagated features of a node also depend on the original feature of the node itself. Multiplications by the inverse degree matrix normalize the propagated feature of nodes such that nodes with higher node degrees do not exhibit larger features and do not influence the propagated features of their neighbor nodes less than nodes with lower node degrees. Notice that the implementation of L layers ensures that the node representation of v_i aggregates over the features of the nodes in its L^{th}-order neighborhood.
In the setting of semi-supervised node classification, Kipf and Welling (2016) train their model by computing the supervised binary cross entropy loss on known node labels. By backpropagation, the loss is used to update the weight matrices W^l in each layer l during training. Stochasticity in the training process is introduced via dropout (Srivastava et al., 2014).

The authors acknowledge the in sample size linearly increasing memory requirement in their full-batch approach. A possible solution would be mini-batch gradient descent. Since propagated node features are convoluted and thus dependent in the conventional GCN framework, simple batching is not possible. For an exact procedure for each mini-batch a model with L layers would take as inputs the adjacency matrix, the feature vectors and the node labels induced by the L^{th} order neighborhood of the nodes in the mini-batch sample. Due to recursive neighborhood expansion, this approach is not feasible for large, dense graphs. Hamilton et al. (2017) propose instead to sample a fixed-size neighborhood for each node in the mini-batch. This procedure, however, results in slow performance for small and sparse graphs compared to the basic GCN framework. An alternative approach based on sampling was proposed later by Chen et al. (2018).

2.2 FastGCN

FastGCNs (Chen et al., 2018) are a fast improvement of GCNs which can also be applied in the inductive setting through the interpretation of graph vertices as i.i.d. samples of some probability distribution and the definition of the loss and each convolutional layer as integrals with respect to vertex embedding functions. Such an interpretation allows for the use of importance sampling to consistently estimate the integrals, which in turn leads to a batched training scheme and reduces memory requirements compared to the conventional implementation of GCNs.
In particular we assume the existance of a graph $G' = (V', E')$ such that V is a set of i.i.d. samples from V' according to the probability measure P. Each layer of the network is interpreted as an embedding function of the vertices

$$h^{l+1}(v_i) := \sigma(\int \hat{A}(v_i, v_j) h^l(v_j) W^l dP(v_j))$$

with $\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ and the loss is thus

$$L = E_{v_i \sim P}[g(h^K(v_i))] = \int g(h^K(v_i)) dP(v_i)$$

for some function g.

These functions are approximated by Monte Carlo. For each layer l, Chen et al. (2018) uniformly sample t_l vertices $u_1^l, ..., u_{t_l}^l$ such that

$$h_{t_{l+1}}^{l+1} := \sigma(\frac{1}{t_l}\sum_{j=1}^{t_l} \hat{A}(v, u_j^l) h_{t_l}^l(u_j^l) W^l), \quad L_{t_0,..,t_L} := \frac{1}{t_L}\sum_{i=1}^{t_L} g(h_{t_L}^L(u_i^L))$$

with $h_{t_0}^0 := h^0$. Chen et al. (2018) show that the estimator of the loss is consistent.

Since the given graph G is already a sample from G', we further need bootstrapping. For each batch, we uniformly sample with replacement at each layer. This is equivalent to sampling the rows of H^l uniformly for each l such that the batch loss is

$$L_{batch} = \frac{1}{t_L}\sum_{i=1}^{t_L} g(H^L(u_i^L, :)), \;\; H^{l+1}(v,:) = \sigma(\frac{N}{t_l}\sum_{j=1}^{t_l} \hat{A}(v, u_j^l) H^l(u_j^l, :) W^{l)}),$$

where the multiplication by N inside the activation function accounts for the normalization of the uniform distributed variables inside the sum. In order to reduce the variance of this sampling approach, Chen et al. (2018) define the probability mass function

$$q(v_i) = \left\|\hat{A}(:, v_i)\right\|^2 / \sum_{v_j \in V} \left\|\hat{A}(:, v_j)\right\|^2, \;\; v_i \in V$$

and sample t_l vertices according to this distribution. The scaling has thus to change such that

$$H^{l+1}(v,:) = \sigma(\frac{N}{t_l}\sum_{j=1}^{t_l} \frac{\hat{A}(v, u_j^l) H^l(u_j^l, :) W^{l)})}{q(u_j^l)}.$$

The corresponding batch gradient may be straightforwardly obtained through applying the chain rule on each H^l. Chen et al. (2018) show that this importance sampling approach results in more accurate performance than the aforementioned uniform sampling procedure.

This way training and testing data can clearly be separated and the algorithm can also be applied in the inductive setting. Further FastGCN reduces the worst case memory requirement to $\sum_{l=1}^{L} t_l$ compared to $|E|$ in the GCN framework. Given input features $H^{(0)}$, the product $\hat{A}H^{(0)}$ in the bottom layer does not change and can be precomputed for the gradient descent approach in order to save computational time.

2.3 Graph Attention Networks

Other extensions to the GCN framework do not focus on reducing computational requirements, but on allowing for additional flexibility. The idea of GATs is to compute the hidden representations of each node in the graph, by attending over its neighbors following a self-attention strategy. While Kipf and Welling (2016) specify the framework of GCNs such that their weights are learned during training, in graph attention networks (Veličković et al., 2018) the weights W^l are defined implicitly over the node features. The resulting fitted model can be applied in the inductive setting, i.e. unseen nodes and even unseen graphs.
In more detail, we define the *attentional mechanism*

$$\begin{aligned} a: \quad & \mathbb{R}^{F^l} \times \mathbb{R}^{F^l} && \longrightarrow \quad \mathbb{R} \\ & (H_i^l, H_j^l) && \longmapsto \quad e_{ij}, \end{aligned}$$

which maps two node features H_i^l and H_j^l to the unnormalized weights e_{ij} by a neural network. Node v_j is further a node in the neighborhood of v_i such that the graph structure is considered in the mechanism. This way Veličković et al. (2018) allow impolitely for different weights of different nodes in a neighborhood. The attention coefficient e_{ij} indicates the importance of node v_j's feature to node v_i. For comparability across neighborhoods, we normalize the weights using the softmax function such that

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{v_k \in \mathcal{N}_i} exp(e_{ik})}.$$

The parameters of the attention mechanism a are trained with the rest of the network. This implementation equips a neural network with the ability to focus on a subset of its inputs.
Similarly to Vaswani et al. (2017), Veličković et al. (2018) stabilize the learning process of the attention mechanism by employing *multihead attention*. Multihead attention implies that the operations of the layer are replicated independently $K \in \mathbb{N}$ differ-

ent times with different parameters, before the outputs are aggregated by concatenation (or averaging for the last layer)

$$H_i^{l+1} = \Big\|_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k W^k H_j^l\right),$$

where α_{ij}^k is the normalized attention coefficient of the k^{th} attention mechanism a^k and W^k the corresponding transformation's weight matrix. If outputs are concatenated, the layer outputs $K \cdot F^l$.

Veličković et al. (2018) further employ dropout according to Srivastava et al. (2014) on the attentional coefficients α_{ij} for regularization especially in small data sets.

In their experiments, Veličković et al. (2018) set the attention mechanism a equal to a single-layer feedforward neural network and LeakyReLU nonlinearity with negative input slop $\alpha = 0.2$. Shang et al. (2018) show that by letting the adjacency matrix to account for weighted or directed edges, a broader class of graphs can be assessed.

3 Experiments

We test the three introduced methods (GCN, FastGCN, GAT) on four benchmarks: (i) classifying academic papers into different subjects using the Citeseer dataset, (ii) classifying Reddit posts as belonging to different communities, (iii) classifying the tags of questions asked on Stack Overflow and (iv) classifying dummy variables on random graphs of different size and node degree. The experiments are conducted in a semi-supervised setting, i.e. we know the complete graph structure from the beginning on but only have sparse label information.

The following subsection summarizes the datasets. Then we introduce our experimental setup and evaluate our numerical results.

3.1 Data

We chose the datasets such that we achieved complexity with respect to structure and volume. For this purpose, three real-world datasets of different areas were analyzed.

Dataset	**Nodes**	**Edges**	**Classes**	**Features**	**Training/Validation/Test**
Citeseer	3,327	4,732	6	3,703	333/998/1996
Reddit	232,965	11,606,919	41	602	23,297 / 69,889 / 139,779
Stackoverflow	23,410	287,808	30	1,000	2,241 / 10,256 / 20,512

Table 1: Dataset Statistics

Citeseer. This citation network downloaded from `https://linqs.soe.ucsc.edu/data` contains over 3,000 Machine Learning papers that can be classified into six research areas. The dataset further contains sparse bag-of-words feature vectors and a list of citation links. We treat the citation links as undirected edges. Similarly to Kipf and Welling (2016) and Yang et al. (2016), we assume a label rate of approximately 10% and let the validation data consist of 30% of the nodes and the test data of 60% of the nodes. This data split ratio is similarly applied to all other datasets.

Reddit. Similarly to Hamilton et al. (2017) and Chen et al. (2018) we predict the community of Reddit posts. Reddit is an online discussion forum where users post and comment on content in different communities called "subreddits". Hamilton et al. (2017) construct a graph dataset from Reddit posts from 50 large communities made in September 2014. The Reddit data is downloaded from `http://snap.stanford.edu/graphsage/`(Leskovec and Krevl, 2014). The node label is the subreddit that a post belongs to and two posts are connected if the same user comments on both. Features are generated by the post's score, the number of comments made on the post and off-the-shelf 300-dimensional GloVe CommonCrawl word vectors for the post title and the average of similar word vectors for all the post's comments.

Stack Overflow. While the two other datasets have been vastly analyzed in literature on GCNs, we introduce a new dataset in this setting. We extracted the first $50,000$ questions posted in 2019 on Stack Overflow, an online questions and answers website for software developers, from `https://data.stackexchange.com/stackoverflow/`. When asking questions, the user is asked to give their post one or several tags related to the topic of the post. For simplicity, we only keep those questions with exactly one tag being among the top 30 tags of the period and aim to predict these tags. We extracted $1,000$ bag-of-words features based on the cleaned text body. Two posts were connected when the same user was active on both. A user is said to be active on a question post if he posed the question, answered it, commented on it, voted on it or edited it.

Simulated data. In order to support some of the observations we made from our analytical study by numerical examples, we generated five graphs of different size and edge density. We generate three random graphs of size $N \in \{10^3, 10^4, 10^5\}$ and node degree $2N$. Edges are assigned uniformly at random. We use one-hot encoded identity features and label all nodes with $Y = 1$ similarly to Kipf and Welling (2016). Additionally, we simulate two random graphs of size $N = 10^4$ and node degrees $\{10N, 50N\}$.

Features of all datasets were row-normalized before training.

3.2 Experimental Setup

We designed our experiments with the goals of providing a rigorous comparison of GCN, FastGCN and GAT with respect to (i) prediction accuracy and (ii) computational requirements.

Inspired by Kipf and Welling (2016), we implement all networks using TensorFlow

(Abadi et al., 2016). In order to speed up computational time, we use sparse representations of the networks inputs. Our implementation was mainly inspired by `https://github.com/tkipf/gcn`, but also `https://github.com/matenure/FastGCN` and `https://github.com/PetarV-/GAT`.

We follow Hamilton et al. (2017) and Chen et al. (2018) and assess the prediction performance based on Micro F1 scores in order to account for imbalanced classes. Since training sizes are relatively small for training on neural networks, we notice high volatility in test results. To surmount this problem, we report mean accuracy and standard deviation for 100 runs and resample training, validation and test data in each run.

Hyperparameter selection. We train all models for a maximum of 50 epochs on the Reddit and Stack Overflow data set and a maximum of 100 epochs for the remaining data sets. In the former case we stop training if the validation loss does not decrease for 5 consecutive epochs, in the latter case early stopping with window size 10 is employed (except for training on random graphs). We use the Adam optimizer (Kingma and Ba, 2014) to minimize the regularized softmax cross-entropy loss function, initializing weights according to Glorot and Bengio (2010). On the random graph datasets, we use a hidden layer size of 16 units and omit dropout and L2 regularization similarly to Kipf and Welling (2016). Since Kipf and Welling (2016) find that an increase in the number of hidden layers and additional residual connections did not improve test accuracies, we implement all networks with two layers as presented in Kipf and Welling (2016); Chen et al. (2018) and Veličković et al. (2018). We chose rectified linear units as activation functions.

Moreover, in order to guard against unintentional "hyperparameter hacking" in the comparisons, we sweep over the same set of hyperparameters for all GCN variants on the real-world datasets choosing the best setting for each variant according to the performance in 5-fold cross-validation. In all settings, we performed hyperparameter selection on the initial learning rate $\{0.01, 0.001, 0.0001\}$, the number of hidden nodes $\{16, 64, 128\}$, the number of attention heads $\{4, 6, 8\}$ for GAT only and the sample size $\{50, 100, 400\}$ for FastGCN only. For simplicity, we equalize the sample size on both layers. The dropout rate was chosen to be 0.5 and the weight decay parameter $5 \cdot 10^4$. For batched training, the batch size was chosen to be 128. For fitting the random graphs we chose that hyperparameter constellation of the real-world dataset with the closest number of nodes. The hyperparameter sets are chosen according to Kipf and Welling (2016) and Chen et al. (2018).

Hardware. Running time is compared on a single machine with 4-core 2.5 GHz Intel Core i7, and 64GB RAM. We also experimented with GPUs, but found that GPUs may not be able to offer major performance benefits compared to CPUs in sparse scenarios for FastGCN (Chen et al., 2018). Kipf and Welling (2016) also note that GPUs often do not meet the memory requirements of GCN.

3.3 Numerical Evaluation

Next to GCN, GAT and FastGCN, we also trained a mini-batched version of GCN. Test results on the real world data sets are summarized in Table 2. Reported numbers denote micro-averaged F1 scores.
Our results are comparable with the results reported by Kipf and Welling (2016), Chen et al. (2018) and Veličković et al. (2018) for similar datasets. Like the latter, we indeed find that the highest test accuracies for each dataset are achieved when using learnable weights as suggested by Veličković et al. (2018). However, the gain is limited to a maximum of two percent points compared to the original GCN framework.
In contrast to Chen et al. (2018), we find that the mini-batched version of GCN gives better results than the basic GCN. Since Chen et al. (2018) do not provide a description of their approach to batched GCN, this difference might be due to different imple-

Method	Citeseer	Reddit	Stack Overflow
GAT	**72.5** $\pm$ 0.9	85.9 $\pm$ 0.9	**82.1** $\pm$ 0.7
GCN	70.2 $\pm$ 0.7	85.8 $\pm$ 0.8	80.2 $\pm$ 0.9
mini-batched GCN	71.9 $\pm$ 0.3	**86.1** $\pm$ 0.2	80.3 $\pm$ 0.6
FastGCN	70.2 $\pm$ 0.4	86.0 $\pm$ 0.3	80.0 $\pm$ 0.5

Table 2: Prediction results as mean of micro-averaged F1 scores in percent over 100 runs for the three real-world datasets followed by the standard deviation in percent after the $\pm$ sign. The best result achieved for each dataset is marked bold.

mentations. Further, we note that GCN often trains for the maximum number of 100 epochs, while batched GCN always stops early. Therefore, we assume that training GCN for more epochs would give higher scores.

Furthermore, we find that FastGCN, although it uses sampling instead of an exact approach, predicts the class of unlabeled nodes, nearly as accurately as GCN. In order to achieve these results, a sample size of 400 was needed. Test accuracy would deteriorate significantly for smaller sample sizes (especially smaller than 100). For example, we achieved an F1 score of only 45.1% and a standard deviation of 4.9% for the Citeseer dataset and a sample size of 50 as can be seen in Figure 1. Although with more samples the per-epoch training time increases, we see in the following that FastGCN is still the fastest method.

Please refer to Table 3 for results on the measured wall-clock training time per epoch (forward pass, cross-entropy calculation, backward pass) for the real world datasets. First of all, GCN is clearly fastest since it is a batched approach and thus updates the model only once per epoch. Although we get the highest prediction performance for GAT, this benefit is compromised by its high training time. This can be explained by the fact that for each layer 8 additional convolutional networks are computed. These convolutional networks further do not accept sparse matrices which additionally increases the computational time compared to the plain GCN framework. So, although

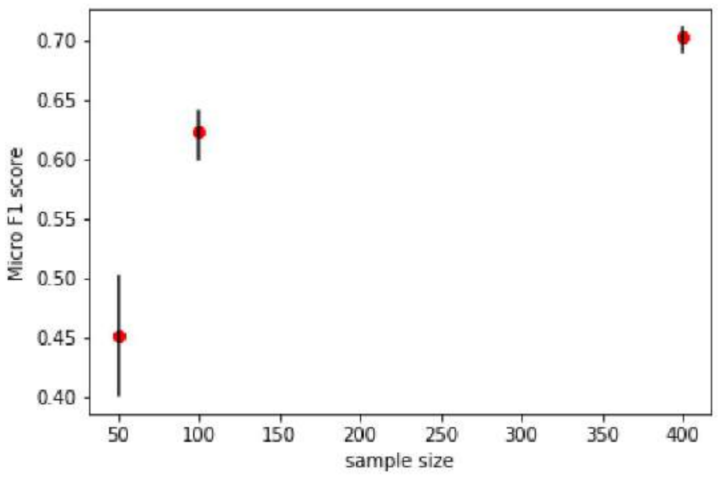

Figure 1: Test micro-averaged F1 score (red dot) against sample size for training FastGCN on the Citeseer dataset. Standard Deviations marked by black vertical line.

Method	**Citeseer**	**Reddit**	**Stack Overflow**
GAT	1.2808	200.6820	18.0298
GCN	**0.0236**	**12.6730**	**0.99823**
mini-batched GCN	0.2043	167.730	8.4251
FastGCN	0.2752	21.4883	2.5377

Table 3: Mean training time in wall-clock seconds over 100 runs per epoch for real-world data sets. The best achieved result for each dataset is marked bold.

the computations of the attentional heads can be parallelized, including attention increases the runtime.

Apparently, batching increases the computation time considerably. Since the batch size is held constant over all datasets, it is clear that the computational time for the batched approaches increases with the number of nodes.

In order to assess the effect of higher node degrees on the mini-batched GCN and FastGCN, we also compare the per batch training time for these methods illustrated in Figure 2. Similarly to Chen et al. (2018), we find that the mini-batched version of GCN is faster than FastGCN for the small dataset. For the other datasets FastGCN out-

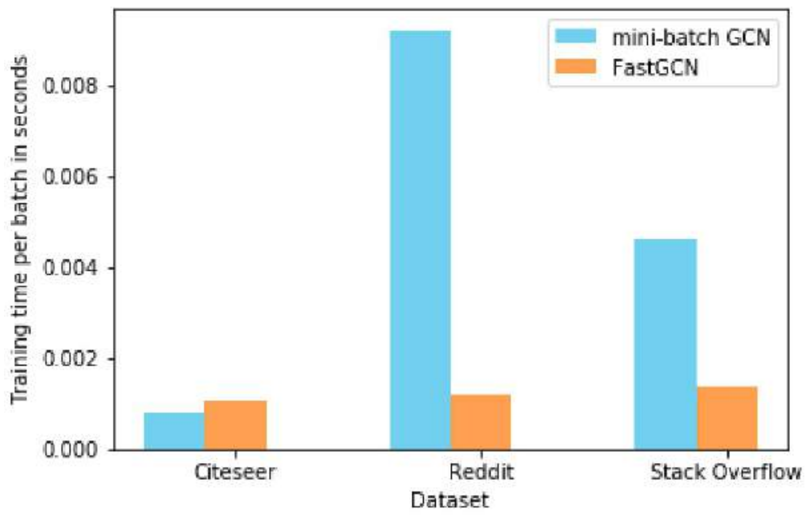

Figure 2: Per batch training time in wall-clock seconds for mini-batch GCN and FastGCN for the real-world datasets.

performs mini-batched GCN with respect to computational time since the sample size of FastGCN remains constant, while higher node degrees result in an exponentially increasing neighborhood expansion. Compared to the results of Chen et al. (2018) the difference between the per-batch training time of FastGCN and mini-batched GCN is smaller in our study. This results from the fact that we regard the transductive setting and not the inductive setting here such that not only GCN but also FastGCN take the whole graph structure as input.

We also notice that the largest driver of the computational time needed for mini-batched GCN is the computation of the second-order neighborhood of the batch nodes. This method is still only implemented for dense matrices. A more efficient implementation could thus considerably boost the performance of this method.

We also report the mean training time per epoch for five simulated graphs which only differ either in size or node degree in Table 4. Such a numerical study enables us to isolate the effect of node degree and size on the training time.

As before, we note that the per epoch training time strongly depends on the graph size. The runtime for GCN and FastGCN do not depend on the node degree, since the for-

Method	$10^3/2\cdot10^3$	$10^4/2\cdot10^4$	$10^5/2\cdot10^5$	$10^4/10\cdot10^4$	$10^4/50\cdot10^4$
GAT	0.3263	14.4899	157.6221	18.7235	25.1502
GCN	**0.0201**	**0.1216**	**0.2798**	**0.2390**	**0.2409**
mini-batched GCN	0.8600	11.1699	874.6443	79.0974	182.3132
FastGCN	1.6736	10.5108	104.3462	11.4104	12.1108

Table 4: Mean training time per epoch in wall-clock seconds over 100 runs per epoch for the random graphs. The columns indicate the size and node degree of the random graph in the form "number of nodes/node degree". The best achieved result for each dataset is marked bold.

mer always trains the whole graph structure, while the latter always sample the same number of nodes for its importance sampling approach.
Even though the original GCN trains faster than all other versions per epoch, it does not scale because of memory limitations. On a single machine with only 16 GB RAM, GCN and GAT ran out of memory for the Reddit and the Stack Overflow dataset. Only FastGCN and the batched version of GCN could be trained then. Further batched GCN converges later than mini-batched GCN such that also the runtime until convergence should be considered.

4 Conclusion

We presented four modifications of GCNs and compared their performance based on eight different datasets. Our study should help to choose the appropriate GCN architecture among the four methods presented. Overall, we found that all methods perform similarly with respect to accuracy on all datasets. While GAT could achieve micro-averaged F1 scores that were up to three percent points over the scores of the basic GCN framework, the training of the attention coefficients increases the training time up to twenty magnitudes compared to the training time of GCN. Batched GCN performed fastest per epoch. The in graph size linearly increasing memory requirement is however a reason why the basic GCN is no viable solution for large graphs. We propose instead to use a mini-batched version of GCN. We find that the recursive neighborhood expansion across layers poses time and memory challenges for training mini-batched GCNs on large, dense graphs. FastGCN runs faster in these cases since the per batch training time does not depend on the node degree or size of the graph.

There are two directions for future work. We found that the benefit of higher prediction accuracy for GATs is compromised by the computational time needed. Since Tensorflow currently does not support sparse inputs for convolutional layers, it would be interesting to implement GAT in PyTorch where such functionalities already exist. Further, the specific setting of this study should be taken into account. While we focused on semi-supervised node classification with sparse label data as in a transductive setting, FastGCN and GAT can also be applied in an inductive setting (i.e. evolving graphs or completely unseen graphs). GAT even allows for directed graphs as input (Veličković et al., 2018). Therefore another comparison study on different GCN methods in the inductive setting could be conducted.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.

Chen, J., Ma, T., and Xiao, C. (2018). FastGCN: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*.

Cucurull, G., Wagstyl, K., Casanova, A., Veličković, P., Jakobsen, E., Drozdzal, M., Romero, A., Evans, A., and Bengio, Y. (2018). Convolutional neural networks for mesh-based parcellation of the cerebral cortex.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Grover, A. and Leskovec, J. (2016). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034.

Hammond, D. K., Vandergheynst, P., and Gribonval, R. (2011). Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150.

Jepsen, T. S. (2018). How to do deep learning on graphs with graph convolutional networks - part 2: Semi-supervised learning with spectral graph convolutions. `https://towardsdatascience.com/how-to-do-deep-learning-on-`

`graphs-with-graph-convolutional-networks-62acf5b143d0`. Accessed: 2019-04-06.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Liberis, E., Veličković, P., Sormanni, P., Vendruscolo, M., and Liò, P. (2018). Parapred: antibody paratope prediction using convolutional and recurrent neural networks. *Bioinformatics*, 34(17):2944–2950.

Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM.

Shang, C., Liu, Q., Chen, K.-S., Sun, J., Lu, J., Yi, J., and Bi, J. (2018). Edge attention-based multi-relational graph convolutional networks. *arXiv preprint arXiv:1802.04944*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018). Graph Attention Networks. *International Conference on Learning Representations*.

Yang, Z., Cohen, W. W., and Salakhutdinov, R. (2016). Revisiting semi-supervised learning with graph embeddings. *arXiv preprint arXiv:1603.08861*.