Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Enhancing Graph Neural Networks with Boosting

Bachelor's Thesis

Nikolas Schäfer

nikolass@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Lukas Faber, Karolis Martinkus
Prof. Dr. Roger Wattenhofer

June 22, 2021

Acknowledgements

I would like to thank my supervisors Lukas Faber and Karolis Martinkus for their continuous support and the opportunity to work on this project.

Abstract

In practice, Graph Neural Networks can often only benefit to a small extent from the informative advantage they have compared to their individual components, edges and features. For instance, in some classification problems, Graph Neural Networks even tend to be outperformed by either edge models or feature models [1]. To tackle this lack of performance, this thesis suggests using the ensemble learner boosting on current benchmark datasets and hereby seeks to establish an alternative to using only Graph Neural Networks. Therefore, this thesis applies state-of-the-art boosting algorithms to Graph Neural Networks and investigates their performance compared to the exclusive use of Graph Neural Networks. We find that boosting Graph Neural Networks offers a small advantage over Graph Neural Networks without boosting, depending on the used architecture and dataset. In some cases, however, they do not represent a considerable improvement.

Contents

Chapter 1

Introduction

Graphs play a central role in representing and abstracting many scientific, societal, and mathematical phenomena and thus help to understand many of the associated complex interrelations, such as social networks or structures of chemical molecules. The abstraction of these problems to a more structural and mathematical level not only eases the visualization for humans but especially enables computers to work with such problems.
Consequently, it is reasonable to assume that deep learning models specialized in graph learning can achieve promising results [2]. Nevertheless, being a relatively new research topic, Graph Neural Networks are still not fully and perfectly developed. One could separate the two building blocks of Graph Neural Networks and graphs - nodes and edges - into two single learning algorithms, one learning only from the node information and the other one only from the edge information. Graph Neural Networks then do not have immense advantages over the combination of their parts [1]. Hence, ways need to be found, so that Graph Neural Networks can benefit more from having both features and edge information. In this regard, this thesis applies different configurations of boosting algorithms with the aim to improve the performance of Graph Neural Networks on node and graph classification, especially to create a clear advantage to using the combination of feature and edge only models. In this thesis, we show that certain Graph Neural Network architectures can be improved on some datasets. Yet, the difference is rather small and, under consideration of the computational cost and the time required, does not always make sense. This thesis is structured as follows: In the first chapter the theoretical background of this thesis is presented together with an explanation of the theory behind Graph Neural Networks and boosting algorithms. After that, we dive into boosting Graph Neural Networks and the conducted experiments and finish with an evaluation of the main results.

Chapter 2

Theoretical Background

2.1 Graph Neural Networks

In this first part, based on the related literature, the fundamental principles of Graph Neural Networks will be discussed. A Graph Neural Network (GNN) is a type of artificial neural network that has become a prominent research topic in the past years. What makes it interesting is the ability to learn from graph data. Graph data are datasets using graphs to display interconnected information. A graph in a mathematical sense is a collection of nodes that are interconnected by edges. These nodes can be understood as data points with a specific informational content revealing properties of the node. The edges, one the other hand, advertise a relation of some sort between the linked nodes. For example, in a graph representing a social network, edges could indicate friendships between people, represented by nodes, that contain information such as a profile picture, age, etc.. Both parts - nodes and edges - contain relevant information for the learning process: Each node has a feature vector which can be seen as information about itself and edges connecting the node to other nodes, meaning the node can also obtain information about his neighbor. The feature vector of each node is converted into an embedding and in several rounds of message passing to each direct neighbor knowledge about the entire graph is expanded to each node. As a result of the training, the GNN can classify nodes and graphs based on information of the nodes' features and their edges [2].
In general, there are three types of classifications a GNN can make. First, node classification which is the most commonly used type. As the name suggests, node classifications try to predict labels of single nodes, e.g. if a user (node) in a social network is a bot [2]. The second type of GNN classification is graph classification where the GNN learns to classify entire graphs. In practice, graph classification is used for example for property prediction based on molecular graph structures [2]. The third type is relation prediction. In this form of classification, the existence and nature of edges are predicted. This of often used for content recommendation in online platforms [2]. In this thesis, however, we focus on the usage of GNNs for node classification and graph classification datasets only.

Knowing the idea and coarse structure of GNNs, we will now dive into how they work specifically.
The defining property of GNNs is the so-called "neural message passing" [2]. This means that node features - here referred to as messages in the form of vectors - are passed between the nodes and iteratively updated using a neural network. In the beginning, each node has a feature vector which is then translated into an embedding. This embedding corresponding to the node is then updated in several rounds of the algorithm according to the aggregated embeddings of the neighboring nodes. The basis of the algorithm is an aggregation function and an update function. The aggregation function collects the embeddings of the neighbors and the update function computes a new embedding from the aggregated embeddings of the neighbors and the old embedding, i.e. from the last iteration. The output of the final layer is the resulting embedding after K iterations of the algorithm. In the course of the iterations, every node obtains more information about the full graph since in the first iteration the new embedding is combined from the embeddings of the direct neighbors, in the second iteration, from the neighbors and neighbors' neighbors, and so on. This way, the nodes end up receiving an embedding unique to the graph and the node information [2]. For instance, the embeddings encode structural information, as well as feature-based information. Structural information is specific to the graph, for example after some iterations of message passing, the embedding could encode information about the degree of the $k-hop$ nodes, whereas feature-based information could be the properties encoded by an embedding that contain information about the neighboring nodes. From a mathematical perspective the purest form of neural message passing is [2]:

$$\mathbf{h}_u^{(k)} = \sigma\big(\mathbf{W}_{self}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{neigh}^{(k)} \sum_{v\in\mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\big) \tag{2.1}$$

where $\mathbf{W}$ is a trainable parameter matrix, σ is a non-linear activation function, and $\mathbf{b}$ is a bias term that is also often omitted in the definition [2]. Originating from this basic definition of a GNN message-passing protocol, many different GNN architectures can be designed, for example by applying a convolutional layer.
The next section introduces the background of this thesis and treats the question as to why boosting GNNs could be beneficial.

2.2 Graph Neural Networks as a Combination of Edges and Node Features

Graphs consist of nodes and edges, i. e. they result from a combination of nodes and edges. This fact - transferred to GNNs - means that they can be seen as a combination of node only and edge only models, thus it is also possible to train only on edges or only on nodes. In this context, the term *node* is replaced by

feature as the nodes contain an information structure that is similar to classical machine learning features. Faber et al. [1] investigate if "prominent GNN problems use both features and edges [...] and to what extent a dataset can be solved with only features or edges" and therefore suggest dividing the training processes of GNNs into training with only features and training only with the information of the graph's edges. In the focus of the analysis stands the question "how well GNNs exploit the combination of features and edges" [1]. This raises the question, whether a GNN is the sum of its parts, or more concretely, how, after training only edges and features separately, the combined solvable set of both, meaning the set of nodes that the two models can learn added together, is related to the solvable subset of a GNN [1]. The single models - feature-only, edge-only, and GNN itself - are described in the next sections.

The architecture of the feature-only model entirely renounces the graph structure of the dataset and by doing so, focuses on training with the features of the nodes. This is implemented by storing the node features in a $node \times feature$ matrix. Then a multilayer perceptron is trained using this resulting feature matrix. This corresponds to training a simple artificial neural network with the belonging weight matrices and nonlinear activation functions. Alternatively, any machine learning algorithm can be used for the feature-only model under the condition that it takes a $sample \times feature$ matrix as input [1].
In contrast to the feature-only model, the edge-only model takes advantage of the graph propagation properties of a GNN. That is to say, the neural message passing - as explained in the section before - is preserved. However, the node features are replaced by all-one vectors. Any other initialization of the embeddings is conceivable as well, but "all-ones" ensures neutrality and flexibility in the training process [1].
As a measurement of the predictive quality of the different models, this thesis suggests to use the solvable set, similar to Faber et al. [1]. Another measurement would be the averaged accuracy of each model. However, this approach lacks consistency and, moreover, misrepresents the solution through a false influence of the number of classes a dataset has, since the accuracy of a model should never be worse than random guessing ($acc \geqslant \frac{1}{K}$, with K being the number of classes). It is on that account, that a solvable set is used to measure the predictive power of a model on a dataset. The solvable set is based on the actual correct predictions following a null hypothesis of the Bernoulli Distribution of the K classes. This means that we have an actual statement about the single correctly classified nodes which are summarized as a mathematical set in the solvable set. Thus, "for the universe of predictions of the dataset P, the solvable set of a model M is:" [1]

$$S(M) = \{p \in P | M \; solves \; p\} \tag{2.2}$$

Based on the analysis of this solvable set, it is then possible to evaluate if

a GNN is needed for a dataset. In particular, if neither features nor edges can predict a majority of the nodes, it could be better to use a GNN.
Intuitively, GNNs combine these two structural components (nodes and edges) of graphs into one end-to-end learning algorithm. They use information about both, graph nodes and graph layout, to learn to make predictions on specific graph properties. Hereby, the GNN can learn only from features or only from edges, due to the flexibility of the algorithm. Thus, having information about the single parts of the graph, in addition to the advantage of disposing of both of them, leads to the hypothesis that GNNs' solvable set is a superset of the solvable set of features and edge models. Mathematically speaking, a GNN is effective if the following expressions

$$\frac{|S(GNN) \cap S(Feautres)|}{S(Features)} \tag{2.3}$$

$$\frac{|S(GNN) \cap S(Edges)|}{S(Edges)} \tag{2.4}$$

have values close to one, whereby circumstances of a nonideal world must be taken into account. This also raises the question about the GNN's ability to predict nodes correctly that neither of the single parts can which is described by the following formula [1].

$$\frac{|S(GNN) \cap U|}{|U|} \tag{2.5}$$

"where U is the set of "unsolved" nodes $U = P\,/(S(Features) \cup S(Edges))$" [1]. This measurement also comes in handy, when comparing boosted GNNs to not boosted GNNs

Faber et al. [1] come to the conclusion that feature and edge models can solve many datasets sufficiently well so that often a GNN is not needed. This applies in particular to node classification. Also, GNNs perform better at predicting features than edges, leaving room for improvement. In particular, for graph classification, GNNs do not always represent the combination of their parts, namely feature and edge classification.
This thesis will now investigate if the performance of GNNs can be enhanced by boosting them with edge-only models. The idea is to compensate weaknesses of GNNs and close the existing gap to enhance GNNs to an end-to-end deep learning model. The concept of boosting is explained in the following chapter.

2.3 Boosting

Boosting is based on the question of whether many weak learners can be boosted (combined) to perform better or at least as good as one strong learner. A weak learner is, generally speaking, a classifier that has an accuracy close to but greater than random guessing. Having K classes, this means that the accuracy is $acc = \frac{1}{K} + e$ with $e \to 0$. A classifier with an accuracy close to 80 or 90 percent on the contrary is considered a strong learner [3].

2.3.1 Binary Classification - AdaBoost

Boosting was first proposed by Freund and Schapire [3] as a sequential learner. They introduced the so-called AdaBoost algorithm which is an application of boosting on binary classification, exemplary so-called decision tree stumps which are decision trees with a depth of one. These, by nature, have an accuracy of at least 50 percent which in the case of two classes corresponds to being better than random guessing. In order to have a detailed understanding of boosting before starting to apply this algorithm on GNNs, it is essential to examine the roots of boosting. AdaBoost is the classical boosting algorithm and will be explained in detail in the next section.
The general idea behind boosting is to improve predictions by training several weak learners, each of which compensates for the weaknesses of its predecessors. This concept is based on the availability of a weak learner, called (weak) hypothesis [3]. AdaBoost proceeds in iterative calls to the base learner hence is trained several times with the same dataset. Generally, AdaBoost is not about minimizing the training error, but rather about setting an upper bound on the training error, for instance, the error is bounded by an exponentially decaying function. Before beginning with the algorithm, a dataset $\mathcal{D}$ must be defined: $\mathcal{D} = (Y_1, x_1), \dots (Y_n, x_n)$ with responses $Y_i \in \mathcal{Y}$ and $x_i \in \mathcal{X}$ $\forall\, i = 1, \dots, n$. In literature, $\mathcal{X}$ and $\mathcal{Y}$ are defined as the domain set and the label (infinite) set respectively and $\mathcal{D}$ is the set of data that can be sampled from $\mathcal{Y}$ and $\mathcal{X}$. Essential for boosting is the definition of a so-called set of hypotheses or hypotheses class: $\mathcal{H} = \{h | h : \mathcal{X} \to \mathcal{Y}\}$ [4]. The AdaBoost formulation that is presented in this paper is almost the exact same one originally defined by Freund and Schapire [3], even though many variants have appeared in scientific literature. The only difference between the original AdaBoost and the one in this paper is that the indicator function is used in the weight updating. The result of this variation is that only the weight of incorrectly classified samples is updated in each iteration, meanwhile the weight of the correctly classified ones is only changed by re-normalizing the entire weight matrix. This modification was made to better align the binary classification algorithm to the multi-class classification algorithm and to then make its origin and analogy to the binary case more visible.

AdaBoost iteratively uses a base algorithm in order to get the hypothesis on the reweighed data and interprets their contribution to create a strong classifier [4]. Essential is the use of sampling weights which in the beginning are equal for all samples. The objective of training is to minimize the error rate weighted by the distribution $D^{(m)}$ on the training set. $\mathbf{Pr}_{i\sim D^{(m)}}$ "denotes the probability with respect to the random selection of an example according to the distribution $D^{(m)}$" [3]. Accordingly, the weighted error rate corresponds to the sum of the incorrectly classified samples.

$$err^{(m)} = \mathbf{Pr}_{i\sim D^{(m)}}[T^{(m)}(x_i) \neq y_i] = \sum_{i=1}^{n} w_i \mathbb{1}(c_i \neq T^{(m)}(x_i)) \tag{2.6}$$

Having specified sample weights then allows training the weak learner a second time with a stronger focus on the samples that were classified incorrectly in the first round. This is in practice realized by increasing the sample weights of exactly those samples and decreasing the correctly classified ones. This process of training the classifier, evaluating the error, and computing new weights to specifically train the error is repeatedly executed until the requested accuracy or convergence is reached. Hereby, the final - ideally strong - classifier is computed as a weighted sum of all the trained classifiers. This particularly means that in the beginning, a number of estimators T has to be set. Moreover, the total number of train data samples has to be set to n. This allows setting the weight of each sample to $w_i = \frac{1}{n}$ for $i = 1, ..., n$, ensuring a distribution of the weights since all sample weights summed up is then equal to on [3]. The following procedure has to be executed for every of the T estimators: First, the classifier has to be fit to the training dataset using the weights. Secondly, the error rate of the classifier on the training data has to be computed, which corresponds to computing the sum of the weights of the samples that were classified wrong. Crucial for boosting is a parameter indicating the quality of a classifier or, in different terms, how sure a classifier is about a certain decision. In AdaBoost, this is realized by alpha which is defined as $\alpha = \log \frac{1-\epsilon}{\epsilon}$ [3]. One can easily see that classifiers with an error rate of 0.5 (random guessing) are assigned an alpha of zero, meaning their "vote" in the final classification does not count. Meanwhile, if the error rate is very small and goes to zero, the alpha increases to infinity, which would stand for a perfect classifier. For the algorithm to work, the weights of the incorrectly classified samples have to be increased. This update of the weights is performed by multiplying the current weights with the exponential of alpha. Finally, the weights have to be re-normalized to keep a distribution by dividing each weight by the sum of all the weights. The strong classifier results from the superposition of all the trained classifiers. Namely, the output for one sample is the class that maximizes the sum of all the alphas of the weak classifiers that produce this class as output [3]. The pseudocode for AdaBoost based on the formulation by Zhu et al. [5] is provided in Algorithm 1.

Algorithm 1 AdaBoost

Given: $(x_1, y_1), ..., (x_n, y_n)$ where $x_i \in \chi, y_i$ in {-1, +1}

1. Initialize: $W_1(i) = 1/n$ for $i = 1, ..., n$
2. For $i = 1, ..., M$:
   (a) Train weak learner $T^{(m)}$ using distribution W_t
   $err^{(m)} = \sum_{i=1}^{n} w_i \mathbb{1}(c_i \neq T^{(m)}(x_i)) / \sum_{i=1}^{n} w_i$
   (b) Compute
   $\alpha^{(m)} = \frac{1}{2} log(\frac{1-err^{(m)}}{err^{(m)}})$
   (c) Set
   $w_i \leftarrow w_i \cdot exp(\alpha^{(m)} \cdot \mathbb{1}(c_i \neq T^{(m)}(x_i))), i = 1, ..., n$
   (d) Re-normalize w_i
3. Output
   $H(\mathbf{x}) = sign(\sum_{m=1}^{M} \alpha^{(m)} T^{(m)}(\mathbf{x}))$

Freund and Schapire [6] showed that the generalization error can be bounded by the training error. With high probability, the generalization error is smaller than:

$$\mathbf{Pr}[H(x) \neq y] + \mathcal{O}\left(\sqrt{\frac{Td}{m}}\right) \tag{2.7}$$

where m is the number of samples, d is the VC-dimension of the weak hypothesis and T is the number of boosting iterations. Hence, in theory, it seems that boosting would overfit when run for too many rounds since a higher T means a higher bound for the error. Yet empirical studies have shown that this, in practice, does not occur, even when performed for more than 1000 rounds. Interestingly, the generalization error even continued to decrease after the training error reached zero. This observation is important for the experiments in this thesis, because similar phenomena happened in the testing process with GNNs [6]. Although AdaBoost was first introduced in the context of binary classification, in this paper a multi-class boosting algorithm is needed. For this purpose, three different approaches for multi-class classification are briefly examined in the following: SAMME, SAMME.R and Adaboost.M1.

2.3.2 Multi-Class Classification - SAMME and SAMME.R

Boosting can be seen as an infinite-dimensional optimization problem. But in the original AdaBoost formulation, no specific loss function was denoted. For SAMME and SAMME.R a specific exponential loss function is defined. SAMME was first formulated by Zhu et al. [5], motivated by finding a natural extension of AdaBoost to multi-class. SAMME stands for Stagewise Additive Modeling using a Multi-class Exponential loss function. The concrete algorithmic implementation of SAMME is very similar to AdaBoost, except for one minor difference: the calculation of *alpha* (Algorithm 2, 2b) is extended by the term $log(K-1)$, K denoting the total number of classes. It can be easily mathematically shown that SAMME reduces to AdaBoost for the case $k = 2$ by inserting into the new equation for *alpha*. Conceptually, this multi-class algorithm has the advantage that the accuracies of the hypotheses do no longer have to be at least 50 percent, but still, random guessing which for multi-class reduced to $acc \geqslant \frac{1}{K}$. As a matter of fact, this is an effect of the new formula for *alpha*. This can be proven by inserting error rates as high as $err \leq 1 - \frac{1}{K}$ in the formula for *alpha* which then is still a positive number. This again is a crucial condition for the algorithm since otherwise the weight of incorrect samples would be reduced instead of increased. This leads to changes in the range of alpha in a way that more weight is put on misclassified data points. Zhu et al. [5] show that the newly added term is not artificial or arbitrary, but the mathematical extension to making SAMME equivalent to fitting a forward stage-wise additive model using a multi-class exponential loss function. The pseudocode for SAMME can be seen in Algorithm 2. It almost strictly follows the formulation by Zhu et al. [5].

Similar in structure to SAMME is SAMME.R. But instead of using the final classification of the hypothesis themselves, SAMME.R uses "real-valued, confidence-rated predictions, such as weighted probability estimates" [5]. It is an ad-hoc boosting algorithm, meaning it uses the computed weighted probability estimates in each iteration. These real-valued estimates give SAMME.R its name (the R stands for Real). The probability estimates of each iterative round are used to derive the weak hypotheses and also state the changes of the sample weights [5]. The concrete SAMME.R formulation is presented in Algorithm 3, based on the formulation by Zhu et al. [5].

A direct and straight-forward approach to transform AdaBoost for binary classification to a multi-class boosting algorithm would be to use the same algorithm but instead of binary classifiers multi-class base learners. It is possible to proceed in the same way, only the output of the weak learner is now $\{0, ..., K\}$, instead of $\{0, 1\}$. The drawback of this boosting algorithm, which was introduced by Freund and Schapire [3] as AdaBoost.M1, is that training error still has to be smaller than 0.5 for the algorithm to work. However, when classifying multiple labels, it is more difficult to achieve such an error, since random guessing is no longer equal to an error rate of 0.5. This is a major disadvantage that

Algorithm 2 SAMME

Given: $(x_1, y_1), ..., (x_n, y_n)$ where $x_i \in \chi, y_i$ in {0,...,K}

1. Initialize: $W_1(i) = 1/n$ for $i = 1, ..., n$
2. For $i = 1, ..., M$:
   (a) Train weak learner $T^{(m)}$ using distribution W_t
   $$err^{(m)} = \sum_{i=1}^{n} w_i \mathbb{1}(c_i \neq T^{(m)}(x_i)) / \sum_{i=1}^{n} w_i$$
   (b) Compute
   $$\alpha^{(m)} = log(\tfrac{1-err^{(m)}}{err^{(m)}}) + log(K-1)$$
   (c) Set
   $$w_i \leftarrow w_i \cdot exp(\alpha^{(m)} \cdot \mathbb{1}(c_i \neq T^{(m)}(x_i))), i = 1, ..., n$$
   (d) Re-normalize w_i
3. Output
   $$C(\mathbf{x}) = \arg\max_k \sum_{m=1}^{M} \alpha^{(m)} \cdot \mathbb{1}(T^{(m)}(\mathbf{x}) = k))$$

AdaBoost.M1 has compared to SAMME or SAMME.R. Consequently it is better to use SAMME than AdaBoost.M1.
Another strategy for multi-class boosting is the "All vs. One" method. Freund and Schapire [3] propose a multi-class boosting algorithm using this reduction from multi-class to binary classification called AdaBoost.MH. Nevertheless, this seems inefficient under consideration that, as a matter of fact, normal AdaBoost already has to be performed for multiple runs. Beyond that, AdaBoost.MH often lacks robustness and, if the posed binary classification task do not have a certain degree of complexity, already one or two wrong classifications by some weak learners suffice to potentially cause a wrong final classification [3]. This means, that either SAMME or SAMME.R seems to be the right choice for boosting GNNs.
In this thesis, we choose to use SAMME.R. The first reason for this is that SAMME.R appears to be the cleaner and mathematically more profound algorithm. Especially, since it uses probability estimates that can express more accurately how sure a hypothesis is about a certain prediction, while in SAMME an arg max value is used instead, which is rounded and therefore does not reflect this aspect as precise as SAMME.R does. Consequently, SAMME.R also incorporates into the prediction, if another label has a high probability to be correct, too. Apart from this, SAMME is very sensitive to an error rate smaller than random guessing. If this is not guaranteed, the weak learner with such an error rate cannot be used. SAMME.R, in contrast, does not depend on good accuracy. This is especially important for this thesis because a GNN might perform worse than random guessing. Finally, Zhu et al. [5] stated that SAMME.R "converges

Algorithm 3 SAMME.R

Given: $(x_1, y_1), ..., (x_n, y_n)$ where $x_i \in \chi, y_i$ in {0,...,K}

1. Initialize: $W_1(i) = 1/n$ for $i = 1, ..., n$
2. For $i = 1, ..., M$:
   (a) Train weak learner $T^{(m)}$ using distribution W_t
   (b) Obtain the weighted class probability estimates
   $p_k^{(m)} = Prob_w(c = k|\mathbf{x}), \ k = 1, ..., K$
   (c) Set
   $h_k^{(m)} \leftarrow (K-1)(\log p_k^{(m)}(\mathbf{x}) - \frac{1}{K}\sum_{k'} \log p_{k'}^{(m)}(\mathbf{x})), \ k = 1, ..., K$
   (d) Set
   $w_i \leftarrow w_i \cdot exp(-\frac{K-1}{K}\mathbf{y}_i^T \log \mathbf{p}^{(m)}(\mathbf{x}_i)), i = 1, ..., n$
   (e) Re-normalize w_i
3. Output

$$C(\mathbf{x}) = \arg\max_k \sum_{m=1}^{M} h_k^{(m)}(\mathbf{x})$$

faster and performs better than same" which is the final tipping point to use SAMME.R in this thesis. A detailed graphical comparison of the three boosting algorithms - AdaBoost, SAMME, and SAMME.R - regarding their convergence of the error rate with the number of iterations is provided in Figure 2.1. The graphical plot illustrates how SAMME.R converges faster than SAMME and after several boosting iterations has a lower error rate.

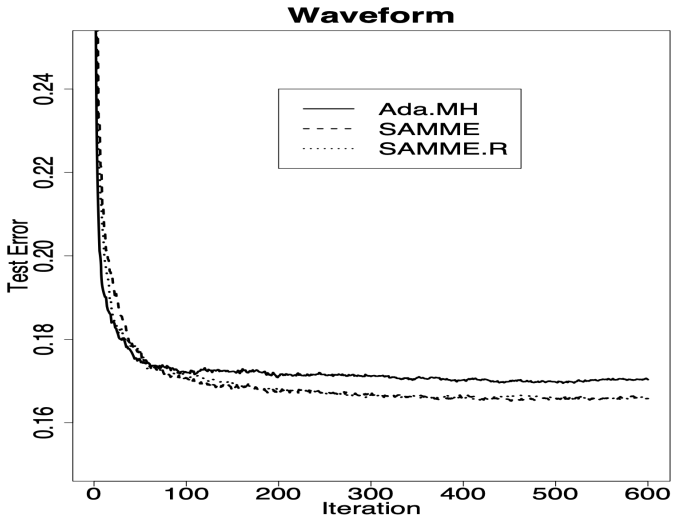


Figure 2.1: Comparison of Adaboost, SAMME and SAMME.R, Figure by Zhu et al. [5]

Chapter 3

Methodology

3.1 Graph Neural Networks with Boosting

This chapter describes the methodical procedure for applying boosting to GNNs. In advance of the implementation of GNNs with boosting, AdaBoost was meticulously investigated and studied by applying it to several machine learning algorithms. This way, it could be tested on simpler applications how well the boosting algorithm works and - most importantly - ensured that the implementation is flawless. At the beginning of the work, boosting was implemented and tested on the most simple use case: decision tree stumps which were implemented with the help of python's Scikit-learn [7] and NumPy [8] libraries. This was followed by a PyTorch [9] AdaBoost application on Logistic Regression. The next logical step was to expand from the binary classification case to SAMME and later SAMME.R on a simple PyTorch Feed-Foward Neural Network. Both in terms of accuracy and convergence similar results to Zhu et al. [5] were achieved. This reassures the decision to use SAMME.R instead of SAMME. Finally, the algorithm had to be applied to GNNs.
Boosting GNNs in principle works just like boosting any other classifier. For training with the multi-class boosting with SAMME.R probability estimates for each possible label are required. This means applying a softmax layer to the output of the GNN giving every class a value between zero and one while the values of all the classes add up to one. Training with sample weights is achieved through the cross-entropy loss. Several GNNs are to be fitted with these sample weights. After training, the weights are updated in a way that increases the weight of incorrectly classified samples. These new weights influence the cross-entropy loss of the next, new GNN that is being trained since larger weights mean a bigger loss and therefore the GNN attaches more attention to the correct classification of these samples.

3.2 Experiments

After having explained boosting and GNNs and therewith setting the theoretical foundations for this thesis, the experimental methodology will be explained in the next section.
The setup of the experiments follows Faber et al. [1] with the same datasets and GNN architectures to ensure consistency and comparability of the results as this thesis seeks to implement the outlook suggested by Faber et al. [1]. In order to get as generally valid and meaningful results as possible, we test some of the most common GNN architectures and investigate their performance on multiple for graph learning designed datasets. The exact setup can be read in [1].
For the implementation of boosting with GNNs, mainly the PyTorch library [9] and the DGL library [10] are used. As for the datasets, we adopt most of the hyper-parameter settings. The maximum number of training epochs is set to 500, but an early stopping mechanism with patience of 25 is implemented in the code. Early stopping occurs frequently, especially in the first few rounds of boosting. Depending on the dataset and GNN architecture in the experiment, the training loss at times goes to zero after several boosting iterations. A possible explanation for the training loss being zero is that the GNN has perfectly learned the train mask of the dataset. As explained priorly, this - what normally is considered heavily overfitted - should not pose a problem with boosting since the generalization error still decreases. We use the Adam optimizer and cross-entropy loss for training. All neural networks have three layers and to obtain probability estimates for SAMME.R, an additional softmax activation function is applied to the output. The embedding width is the maximum of twice the number of inputs or outputs but bounded to 128 [1]. We set the number of boosting iterations to ten. This number ensures that we already see some changes on the solvable set but at the same time have a reasonable training time.
Finally, all configurations - of which there are three - are run with ten different seeds. The first configuration is to train all GNN architectures without boosting with all the datasets and hereby setting a baseline in terms of accuracy and the solvable set for comparison with the other configurations. The second configuration is to boost all GNN architectures with all datasets and ten boosting iterations. It is expected that this configuration already leads to the augmentation of the solvable set. Lastly, a modified boosting configuration is used: In each of the ten boosting iterations another model is randomly used for training: with equal probability, it is either the case that the specific GNN for that boosting is trained, a feature-only model, or an edge-only model. This is the suggested extension of GNNs and their parts by Faber et al. [1] and is here referred to as mixed boosting. Then statistical tests are performed on the test predictions with a 99.9 percent confidence level.

Chapter 4

Main Results

The results of training the three described GNN boosting configurations were evaluated according to the benchmark of the respective solvable set. Thereby two aspects are particularly considered: the solvable set for each configuration itself (no boosting, boosting, mixed boosting) and each dataset as well as what the predictive advantage of a configuration over another is, that is to say, what predictions that GNNs without boosting cannot make correctly, can boosting or the mixed boosting approach make additionally. It is important to also test this the other way around, meaning what advantage GNNs without boosting do have over the boosting configurations. This helps to avoid a one-sided perspective on the results.
The values in Table 4.1 represent the mean count of runs that pass the statistical test that was executed for testing ($p = 0.001$ confidence level). In Table 4.1 we can see that the normal boosting approach with SAMME.R generally works better than ensembling GNNs, edge-only, and feature-only models since the scores for normal boosting on the solvable set are always at least as good as for mixed boosting. Compared to only GNNs without boosting, there is a tendency that boosting performs slightly better than no boosting, except for the Mutag, AMZN-Photo, OGBN-Arxiv, and Reddit-B dataset. Even though there are improvements through boosting of up to three percent, we can also observe drastic deteriorations, for example for REDDIT-B. This suggests the assumption that some datasets are more suitable for boosting algorithms than others.

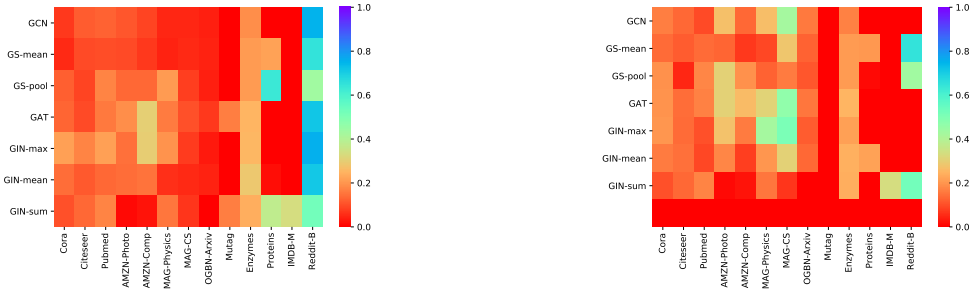
In addition to the accuracy as evaluation metric, we take into account how the actual solvable set changes through boosting, i. e. which correct predictions GNNs with boosting can make that non-boosted GNNs cannot make. Figure 4.1 illustrates the share of predictions a GNN cannot solve that boosting can solve and what percentage of predictions that GNNs do not solve, is solved by mixed boosting. Figure 4.2, in contrast to that, shows what percentage of predictions that the two boosting approaches cannot solve, is solved by a simple GNN. These figures help to assess to what extent the boosting methods increased and changed the solvable set. We observe that the boosting configurations can often solve around 10 to 30 percent of the predictions not solved by a GNN, depending on the particular dataset and the GNN architecture used for boosting. In Figure 4.1,

Dataset	No Boosting	Mixed Boosting	Normal Boosting
Cora	0.782	0.744	0.784
Citeseer	0.64	0.634	0.659
Pubmed	0.743	0.72	0.758
AMZN-Photo	0.891	0.877	0.882
AMZN-Comp	0.776	0.745	0.779
MAG-Physics	0.944	0.945	0.945
MAG-CS	0.921	0.93	0.921
OGBN-Arxiv	0.598	0.561	0.575
Mutag	0.45	0.0	0.4
Enzymes	0.2	0.233	0.233
Proteins	0.562	0.357	0.598
IMDB-M	0.04	0.333	0.333
Reddit-B	0.625	0.485	0.485

Table 4.1: The three columns "No Boosting", "Mixed Boosting", and "Normal Boosting" show the prediction score resulting from performing a statistical test with 99.9 percent confidence level on the solvable set of ten runs with different seeds for the respective dataset. "No Boosting" refers to the classical training of a GNN without boosting. "Mixed Boosting" contains the values for the configuration in which we randomly use a GNN, an edge-only model, or a feature-only model. "Normal Boosting" shows the values for boosting GNNs with the normal SAMME.R algorithm. The first eight datasets are node classification datasets and the last five are graph classification datasets.

we observe that normal boosting performs slightly better than mixed boosting. It is remarkable that normal boosting can solve a big share of the graphs that were incorrectly classified by simple GNNs on the REDDIT-B dataset.

On the other hand, when it comes to the predictions that can be solved by a simple GNN, but not by the boosting configurations, the image turns out differently: the share is often in the range of 30 to 70 percent, both for the comparison with normal boosting and mixed boosting. This is particularly applicable to the graph classification datasets. Interestingly, for the node classification datasets the GCN, GS-mean, GS-pool, and GAT architectures can hardly solve any of the predictions that normal boosting cannot solve. Meanwhile, the GIN-sum architecture can solve a big amount of predictions that neither of the two boosting configurations can.
So, we can state that boosting helps solving some of the predictions that could not be solved without boosting. However, at the same time boosted GNNs can no longer solve predictions that a normal GNN can. It seems that rather a shift than an extension of the solvable set takes place that is caused by a stronger focus on correctly classifying originally not solved predictions and an associated



(a) (b)

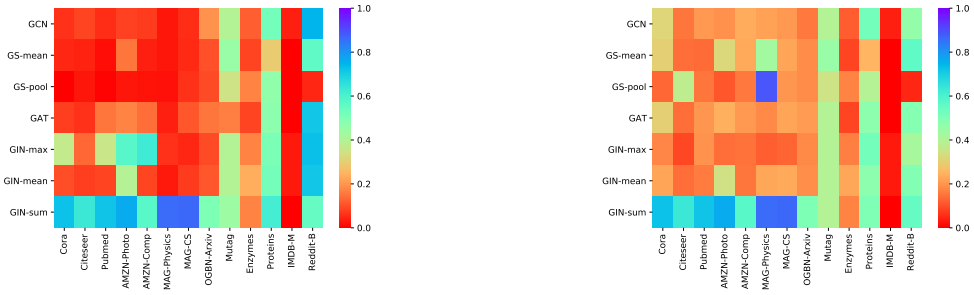
Figure 4.1: What percentage of predictions that a GNN cannot make correctly can be made by GNNs with normal boosting (a) or by GNNs with mixed boosting? This can be a measurement of to what extent boosting leads to an extension of the solvable set. Higher values mean that the boosting approach can solve a big ratio of the predictions that only GNNs cannot solve. (Higher values are good).

neglect of previously solved predictions.

Lastly, Faber et al. [1] state that the considered GNN architectures perform very similarly, in the sense that they all have a solvable set with a large intersection. We want to examine if this is also accurate for boosted GNNs. Figure 4.3 shows the Jaccard similarity of the different GNN architectures across all datasets for normal boosting and mixed boosting. It is directly noticeable that the Jaccard similarity between all the architectures and GIN-sum is zero while the rest has rather good values, confirming that the similar performance of different GNN architectures is preserved for both boosting configurations. Nevertheless, there seem to be GNNs that work better for boosting than others, for example, GAT.

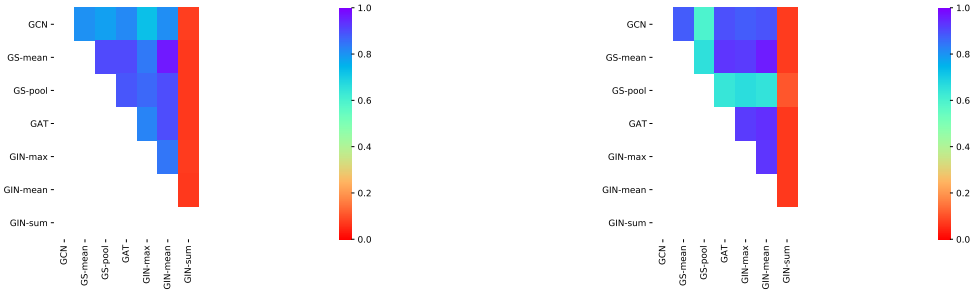
In summary, boosting GNNs can not meet the expectations to represent a clear improvement of GNNs without boosting and also do not enhance their performance in a way that suffices to have an absolute advantage over edge-only and feature-only models. Only assumptions can be made about why this is the case. One possibility could be that GNNs do not have perfect prerequisites to be boosted because of their structure and learning algorithm. This would mean that a machine learning model that is build differently would benefit more from boosting on the same datasets. To investigate this assumption the effect of boosting on a multilayer perceptron has been examined using the same experimental setup as for GNNs. Table 4.2 indicates that on average the accuracy after ten boosting iterations of a multilayer perceptron increases more than the accuracy of boosted GNNs on several exemplary datasets, which would indeed suggest that GNNs are not necessary suitable for boosting.

Another possible explanation for the poor effect of boosting could be that the



(a) (b)

Figure 4.2: Similarly to Figure 4.1: What percentage of predictions that a normal boosting (a) or mixed boosting (b) cannot make correctly can be made by GNNs without boosting? This shows how many predictions boosting approaches "lose" for the sake of making predictions a normal GNN cannot make. Higher values mean that the GNNs without boosting can solve a big ratio of the predictions that the boosting configurations cannot solve. (Lower values are good).



(a) (b)

Figure 4.3: What are the predictive differences of the GNN models? This figure shows the Jaccard similarity of solvable sets of all the GNN architectures with (a) normal boosting and (b) mixed boosting over all datasets. Higher values mean that two GNN architectures can solve very similar nodes.

Dataset	No Boosting	Normal Boosting
Cora	0.419	0.498
Citeseer	0.438	0.481
AMZN-Photo	0.669	0.713
AMZN-Comp	0.444	0.537

Table 4.2: Similar to Figure 4.1: The columns "No Boosting"and "Normal Boosting" show the prediction score of a multilayer perceptron resulting from performing a statistical test with 99.9 percent confidence level on the solvable set of ten runs with different seeds for the respective dataset. "No Boosting" refers to the classical training of a multilayer perceptron without boosting and "Normal Boosting" shows the scores for boosting a multilayer perceptron with the normal SAMME.R algorithm. The four depicted dataset clearly demonstrate the effect of boosting.

GNNs have already almost reached convergence before boosting. Having a look at Figure 2.1 we can see that the error rate, and thus also the accuracy, converge after many iterations. The accuracy of a model that per se is rather a strong learner will not increase exponentially but converge after few iterations since it already has a good error rate. If the GNN has a good accuracy this would mean that the effect of boosting is not as strong as if it was a weak learner.

Chapter 5

Conclusion

GNNs do not necessarily benefit from their informative advantage through the information they have about node features and graph structure compared to machine learning models that only use features or edges. This thesis applied the multi-class boosting algorithm SAMME.R with the objective to enhance the performance of GNNs in a way that they offer a clear advantage over their parts, features and edges. Two different boosting configurations were implemented: First, a normal boosting approach that follows the algorithm SAMME.R by Zhu et al [5]. and always uses the same GNN architecture as a weak learner. Secondly, a boosting mixture, that randomly uses either a specific GNN architecture, only its features, or only its edges and thereby ensembles GNNs with their parts.
We come to the conclusion that the normal boosting approach performs better than mixed boosting. Yet, none of the two configurations represents a clear or consistent improvement over the use of GNNs without boosting. Despite the fact that normal boosting achieved worse accuracies on some datasets than just GNNs, boosting can be seen as a valid alternative depending on the dataset and the GNN architecture. Whether to use boosting or not is also a cost-benefit consideration since there is a rather small improvement with significantly higher computational effort. That is why it has to be decided situationally if boosting a GNN makes sense. The fact that in this thesis boosting is only performed with ten iterations should not be neglected when assessing the results since with more iterations a stronger effect of boosting is conceivable.
The reasons why boosting does not achieve as good results with GNNs as with other machine learning models such as decision trees or a multilayer perceptron are unclear at this point and can be subject of further research. This can also include the consideration of whether another ensembling technique such as bagging could achieve better results.

Bibliography

[1] L. Faber, Y. Lu, and R. Wattenhofer, "Should graph neural networks use features, edges, or both?" Mar. 2021. [Online]. Available: https://arxiv.org/abs/2103.06857

[2] W. L. Hamilton, "Graph representation learning," in *Synthesis Lectures on Artificial Intelligence and Machine Learning, Vol. 14, No. 3*, 2020, pp. 1–159.

[3] Y. Freund and R. E. Schapire, "Boosting: Foundations and algorithms," in *Adaptive Computation and Machine Learning Series*, 2012, pp. 1–528.

[4] A. Ferrario and R. Hämmerli, "On boosting: Theory and applications," 2019. [Online]. Available: https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/383242/20190611_Boosting_Chapter_FINAL.pdf?sequence=1&isAllowed=y

[5] J. Zhu, S. Rosset, H. Zou, and T. Hastie, "Mutli-class adaboost," 2006. [Online]. Available: https://web.stanford.edu/~hastie/Papers/samme.pdf

[6] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of online learning and an application to boosting," in *Journal of Computer and System Sciences, 55*, 1997, pp. 119–139.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style,

high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

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