Learning Discrete Structures with Graph Neural Networks

Andreas Grafberger
Data Analytics and Machine Learning Group
Technical University of Munich
andreas.grafberger@tum.de

April 8, 2020

Abstract

In recent years the field of Graph Neural Networks has emerged and seen tremendous interest. Especially as it is now possible to use modern deep learning methods for graphs, these breakthroughs opened up a new world of potential applications. Many popular Graph Neural Networks, however, rely on existing, high-quality graph information. In their 2019 paper, Franceschi et al. have presented a novel model that jointly learns both the parameters of such a model, as well as the structure of the graph itself. It achieves state-of-the-art in various tasks. Unfortunately, their proposed method does not scale to large graphs as its number of learnable parameters grows quadratically with the size of the graph. Our proposed model, which is based on the Graph Auto-Encoder, modifies their approach and learns a discrete graph distribution by learning node embeddings only, therefore scaling linearly instead of quadratically. We also present two methods to sparsify our sampled graph effectively. Although we find that our method does not perform as well as the method of Franceschi et al., it still might be useful in cases where their model can not be used for scalability reasons. We see it as a starting point for further research into similar models and draw a comparison to similar work that was recently published.

1 Introduction

Relational Learning and *Graph Neural Network* (GNN) research specifically has seen tremendous interest in the last few years. These new type of neural networks have been successfully applied to computer vision problems[12], transductive classification[19], link prediction[18] and more[1]. However, the performance of these methods strongly depends on the available graph information, it for examples has been shown that noise in the input graph can cause such models to fail or perform poorly.[32] GNNs like the *Graph Attention Network*[26] fix some of these shortcomings but are limited in various ways; they can not add completely new edges and do not allow working with discrete distributions over graphs, which inherently provides more freedom to the model and comes with other benefits. To fix this shortcoming, Franceschi et al. recently published a method called *LDS*[11], that can simultaneously learn the parameters of a GNN, as well as a graph distribution that is optimal for the task at hand. For each potential edge in a graph, they explicitly learn a separate, independent Bernoulli parameter that represents the probability that two nodes should be connected. To classify the nodes in a graph, during training and testing, they sample a discrete graph from the learned graph distribution and feed it to a GNN that performs a down-stream task like node classification. Their method not only outperforms previous approaches that only learn the parameters of a GNN, but it is more resilient to noisy graphs or even proves to be useful when no initial graph structure is available. However, their method does not scale to large graphs as the number of learnable parameters of their graph distribution grows quadratically with the number of nodes in the graph. In this work, we build on theirs and propose an al-

ternative graph distribution that aims to provide better scalability while still maintaining comparable performance. The model we present here is based on the *Graph Auto-Encoder*[18]. Instead of learning individual edge probabilities explicitly, we train a separate GCN that produces node embeddings. The probability that two nodes should be connected with an edge is calculated based on the distance of their calculated embeddings. Moreover, various design decisions such as how we enforce sparsity in our graph distribution are explained in detail. We find that our method does not perform as well as LDS but still significantly outperforms a standard GCN for two common graph datasets. We also present related work that was published in the meantime and briefly compare it to ours.[4, 31]

2 Background

The following chapter contains a summary of the notations used throughout this work and a brief explanation of the machine learning models we build upon.

2.1 Graph Notation

We refer to a graph G as a tuple (V, E) where $V = \{v_1, ..., v_n\}$ is a set of n nodes and $E \subseteq V \times V$ a set containing (v_i, v_j) if there exists an edge from v_i to v_j. This graph can also be expressed as an adjacency matrix $A \in \mathbb{R}^{n \times n}$. In this work, we exclusively consider unweighted graphs, therefore $A \in \{0, 1\}^{n \times n}$ and $A_{i,j} = 1 \iff (v_i, v_j) \in E$ and $A_{i,j} = 0$ else. As in [19] and [11] we mostly assume the graph is undirected, so $A_{i,j} = A_{j,i}$ if not otherwise stated. Whenever there exist features for the nodes in a graph they are represented as $X \in \mathbb{R}^{n \times m}$ where m is the dimensionality of the features. For node classification problems each node belongs to exactly one class, represented for the whole graph as a vector $Y \in \{C_1, C_2, ...\}^n$. Various models in this work do not operate on the adjacency matrix A directly but on a normalized version $\tilde{A} = \hat{D}^{-\frac{1}{2}} \hat{A} \hat{D}^{-\frac{1}{2}}$. $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix representing the node degree $D_{i,i} = \sum_j A_{i,j}$ and the operator $\hat{}$ adds self-loops to a matrix ($\hat{U} = U + I$). For easier readability, we continue to differentiate between A and $\tilde{A}$ throughout this work.

2.2 Graph Neural Network

Graph Neural Networks are a family of machine learning methods that are especially well-suited for relational learning and there exist a variety of possible formulations. For an extensive overview the interested reader is referred to [29] and to [1] for a thorough motivation. In this work, we concentrate on the *Graph Convolutional Network* (GCN) as presented in [19] and explain it in the following part. Fundamentally a GCN comprises simple feed-forward neural network layers but allows nodes in a graph to exchange local information (often referred to as messages). In successive time-steps each node aggregates (usually the sum or average) the features of all connected nodes and then applies a linear transformation to each node's updated embedding, followed by a non-linearity (here ReLU). With each successive time-step information thus gets further distributed over the graph in a one-hop neighborhood. To perform node classification, in the final round of message passing the linear transformation is followed by a softmax to calculate normalized class probabilities. Note that the parameters for the linear transformations are shared across all nodes but usually not over successive time-steps. A GCN that performs two rounds of this process (also often referred to as *message passing*[12]) can be written as

$$f_w(X, A) = \text{softmax}\left(\tilde{A}\,\text{ReLU}\left(\tilde{A}XW^{(0)}\right)W^{(1)}\right) \quad (1)$$

Before applying each activation function, a learned bias vector is added to the updated embeddings but we omit it for easier readability. When using a GCN to classify nodes in a graph this boils down to solving the following optimization problem:

$$w^* \in \underset{w}{\text{argmin}}\ \mathcal{L}\left(f_w(X, A), Y\right) \quad (2)$$

where $\mathcal{L}$ is some classification loss like the *categorical cross-entropy*. Note how no model parameters operate on the adjacency matrix directly and that in no way the model can re-weight existing connections or add new edges. This has the consequence that the classification performance strongly relies on high-quality graph information.

2.3 Graph Auto-Encoder

Building on GCNs, [18] introduced two variants of *Graph Auto-Encoders*, models that learn latent

representations for nodes in a graph and that can also be used as graph generative models. Note that here we only focus on the deterministic and not the probabilistic variant of their model. Like a standard *Auto-Encoder*[17] their model comprises an *encoder* and *decoder* component. The **encoder** is a GCN that learns an embedding for each node and can be written as

$$Z = f_w(X, A) = \tilde{A}\,\mathrm{ReLU}\left(\tilde{A}XW^{(0)}\right)W^{(1)}. \quad (3)$$

We also omit the bias here for easier readability. It should be noted that the model does not require the existence of node features X but the authors show in their work that incorporating them, if they are available, greatly improves performance. In such a case where X is undefined, we can set it to the identity matrix and $XW^{(0)}$ then amounts to directly learning an initial embedding for each node. The task of the **decoder** is to reconstruct the original adjacency matrix A by computing a pairwise dot-product of node embeddings

$$p\left(A_{ij} = 1 | \mathbf{z}_i, \mathbf{z}_j\right) = \sigma\left(\mathbf{z}_i^\top \mathbf{z}_j\right) = \sigma\left(ZZ^\top\right)_{ij}. \quad (4)$$

As the decoder has no learnable parameters we can optimize a graph distribution simply by searching for

$$w^* \in \underset{w}{\mathrm{argmin}}\ \mathcal{L}\left(\sigma\left(ZZ^\top\right), A\right), Z = f_w\left(X, A\right) \quad (5)$$

where $\mathcal{L}$ is the weighted *binary cross-entropy*. New graphs can be generated with $A^* \sim \mathrm{Ber}\left(ZZ^\top\right)$.

3 Related Work

We already pointed out that the method presented in [11] suffers from severe scalability problems. But before we present our potential improvements in the next section, this one contains a more detailed explanation of their method and also presents a selection of the most similar work that has been published in the time we worked on ours. For completeness we also advise the reader to look at [15] and [24] as these publications follow similar ideas but focus on slightly different problems.

3.1 LDS

In *Learning Discrete Structures for Graph Neural Networks*[11] (LDS), Franceschi et al. argue that in various problem settings, such as when no graph structure is available or whenever the given graph information is imperfect, one could rephrase the problem entirely. Instead of only optimizing a GCN's parameters as presented in equation 2, one can define a distribution over graphs P_θ and minimize a loss w.r.t. both the parameters of the GCN and those of the graph distribution together. This can be expressed as

$$\min_\theta \min_w \mathbb{E}_{A\sim P_\theta}\left[\mathcal{L}\left(f_w(X, A), Y\right)\right]. \quad (6)$$

To optimize this equation they build on their earlier work [9] and treat the graph distribution parameters as hyper-parameters in a *Bilevel-Programming*[5] problem. Rephrasing equation 6 as a Bilevel problem boils down to splitting the equation into an *inner* and *outer* objective. The inner objective is responsible for finding GCN parameters that optimize an empirical training loss given a fixed graph distribution. The outer objective aims to optimize the parameters of the graph distribution by minimizing an empirical loss on a separate validation set, assuming the parameters of the GCN are optimal w.r.t. the inner objective. Written as a formula the final optimization problem is

$$\boxed{\min_{\theta\in\overline{\mathcal{H}}_N} \mathbb{E}_{A\sim\mathrm{Ber}(\theta)}\left[\mathcal{L}_{val}\left(w_\theta, A\right)\right]\ \textit{Outer Problem}}$$
$$\boxed{\text{s.t. } w_\theta = \arg\min_w \mathbb{E}_{A\sim\mathrm{Ber}(\theta)}[\mathcal{L}_{train}(w, A)] \quad \textit{Inner Problem}} \quad (7)$$

$\overline{\mathcal{H}}_N$ is the convex hull of all possible graph distribution parameters. The authors refer to the corresponding training algorithm as *structure learning via hyper-gradient descent*. Optimizing the inner problem can easily be done with any variant of gradient descent, as during training $\nabla_{w_t}\mathcal{L}_{train}$ only depends on the current set of GCN parameters w_t at each training step t. However, computing $\nabla_\theta\mathbb{E}\left[\mathcal{L}_{val}\right]$ is a bit more involved.

In each gradient-descent iteration of the inner problem a new sampled adjacency matrix $A \sim \mathrm{Ber}(\theta)$ is used with the GCN to calculate $\nabla_{w_t}\mathcal{L}_{train}$. When the final GCN parameters w_t are then used to calculate $\mathcal{L}_{val}\left(w_t, A\right)$, the gradient $\nabla_\theta\mathbb{E}\left[\mathcal{L}_{val}\right]$ has a dependency on w_i for $i \in \{1, ..., t\}$, so across all weight updates of w. This setup is very similar to

Backpropagation Through Time[27] which is used to train recurrent neural networks. The way this can be optimized is by unrolling the optimization graph of the training dynamics of the inner problem, which can easily be done with modern frameworks for automatic differentiation. In practice, only the last λ (which is treated as a regular hyper-parameter) optimization steps are actually unrolled due to computational limitations. For details on how the authors implement the optimization please refer to their work.

One more notable design decision is how the term $\nabla_\theta A$ is computed, as it appears multiple times when calculating $\nabla_\theta \mathbb{E}\left[\mathcal{L}_{val}\right]$. As we sample from a discrete distribution, calculating that gradient is non-trivial. Instead of using algorithms like *REINFORCE*[28] to estimate the gradient, the *straight-through estimator*[2] is used by setting $\nabla_\theta A := I$. This also allows them to not use a differentiable approximation of the Bernoulli distribution like the *Gumbel-Softmax*[14] but to directly use samples drawn from a discrete Bernoulli distribution. Although this gradient is biased, this approach allows for easier optimization because the gradients can just flow directly through the sampling step. In their experiments Franceschi et al. demonstrate that LDS outperforms other baselines on a variety of problems. It shows especially promising results in scenarios where either no prior graph information is available or when it is noisy. We will describe their experiment setup and results later in this work where we also compare it to ours.

3.2 GRCN

With a similar idea in mind [31] presents the *Graph-Revised Convolutional Network* (GRCN). Just like we do, they separate the GCN responsible for node classification and the module that predicts the graph structure. Their *graph revision module* however is non-probabilistic and the calculated adjacency matrix is directly fed into the classification module. The graph revision module can be written as

$$\begin{aligned} &\text{GRCN}(A, X) = GCN_c(\widetilde{A}, X) \\ &\widetilde{A} = A + K\left(GCN_g(A, X)\right) \end{aligned} \tag{8}$$

where GCN_c is the GCN for node classification, GCN_g the graph revision module and $K(Z) := ZZ^\top$ some kernel function, here chosen to be the dot product. Given the computed adjacency matrix they additionally thin out the graph by only allowing connections for each node to the k-nearest-neighbors. This allows them to retrieve a sparse graph and use sparse matrix operations, making running the model very efficient. However, gradients only flow through the non-zero values of the adjacency matrix. They directly optimize both model components by minimizing a common training loss only, making their optimization much easier. For details we refer to their paper, a short comparison with our model and LDS follows in the experiments section 6.

3.3 DIAL-GNN

Deep Iterative and Adaptive Learning for Graph Neural Networks (DIAL-GNN)[4] is another publication that presents a model that jointly learns graph structure and embeddings that can be used for downstream tasks like node classification. Their two primary contributions are a) treating graph structure learning as a similarity metric learning task and b) a novel iterative improvement mechanism for more efficient learning. They also use three different regularization terms to introduce priors like sparsity or smoothness for the graph. To predict if nodes i and j with features $\vec{x}_i, \vec{x}_j$ respectively should be connected, they learn m similarity measures $s_{ij}^k = cos(\vec{w}_k \odot \vec{x}_i, \vec{w}_k \odot \vec{x}_j), k \in \{1, ..., m\}$ where $\vec{w}_k$ are the learnable parameters. To receive the final adjacency matrix they average the individual similarity scores $s_{ij} = \sum_{k=1}^{m} s_{ij}^k$ and then threshold that result to get a sparse graph via

$$A_{ij} = \begin{cases} s_{ij} & s_{ij} > \varepsilon \\ 0 & \text{otherwise} \end{cases}$$

. ε is a hyper-parameter. Like GRCN they therefore do not follow a probabilistic approach which allows them to optimize the downstream classification GCN and the similarity function by simply minimizing a (regularized) common classification loss. For details on the iterative refinement please see their paper.

4 Our Contribution

Inspired by the GAE explained earlier in 2.3 we modify LDS by parameterizing the graph distribution differently. All other elements of the optimization

such as the formulation of the Bilevel problem and using a separate GCN that, given a sampled graph from our distribution performs the actual downstream task of node classification, stay the same. We still sample from a Bernoulli distribution but the individual parameters come from a GAE:

$$\begin{aligned} Z &= GCN_g(X, A) \\ \widehat{A} &\sim \text{Ber}(\sigma(ZZ^T)) \end{aligned} \tag{9}$$

We additionally pre-train GCN_g by training it to reconstruct the original adjacency matrix A. This makes it easier for the model in the beginning of the training and allows us to introduce prior knowledge about the graph structure, if it exists.

Naively using this solution however does not work and we had to make various modifications which are explained in the rest of this section.

4.1 Graph Sparsification

Directly using the sampled adjacency matrix $\widehat{A}$ from above and feeding it into a downstream classification, GCN lead to poor results in first experiments. As the values of the node embeddings and therefore also their dot product is centered around 0, passing them through the sigmoid leads to Bernoulli parameters that are ≈ 0.5 everywhere. Any graph sampled from that distribution is too connected. We, therefore, need to sparsify the graph in a post-processing step. This is also done in similar work like [31, 4]. Initially we tried adding a power to the Bernoulli parameters, therefore sampling from

$$\widehat{A} \sim \text{Ber}(\sigma(ZZ^T)^n), n > 2.$$

All probabilities are thus artificially decreased additionally. Although this worked in first experiments, it leads to unstable gradients and we, therefore, discarded it. The two methods that have proven to be useful are the following:
Learning an **affine transformation**

$$\text{Ber}(\sigma(aZZ^T - b)) \text{ for } a, b \in \mathbb{R}$$

for the node similarity values before passing them through a sigmoid is one of them. a and b are simply learned additionally via gradient descent during training. This way the model can modify the slope of the sigmoid and by subtracting the offset b, learn a soft threshold for embedding similarities before they lead to too large probabilities.
For each node in the graph, we also restrict the sampled edges to belong to a **k-nearest-neighbor graph** based on the node embeddings. We therefore only set the entry in the adjacency matrix for a node-pair to 1 if their learned embeddings are close. Because we use a straight-through estimator to calculate gradients for the backwards pass during backpropagation, we can still get gradients for all values in the adjacency matrix although only a very small portion of them are non-zero. Instead of restricting samples to be from a k-nearest-neighbor graph we also tried tuning a hard threshold of the embedding similarities but that always performed worse than the former method.

4.2 Model Variants

Now follows a short collection of various tweaks and model variants that we tried in our experiments and most of them were exposed as regular hyperparameters for our final grid-search. A first modification was to add the original adjacency matrix to our calculated embedding similarity matrix. This is also done in [31] but as we need the parameters of the Bernoulli distribution to be between 0 and 1, they are manually cut off if they are outside that range.

$$\begin{aligned} Z &= GCN_g(X, A) \\ \widehat{A} &\sim \text{Ber}(clamp(A + \sigma(ZZ^T))) \end{aligned}$$

We also tried to learn an embedding vector for each node directly without the use of a GCN but did not achieve competitive results. We shortly also experimented with hyperbolic embeddings like explained in [21] instead of using the euclidian embeddings from a GCN but did not achieve satisfactory results in preliminary experiments and stopped exploring this direction. We nonetheless think that this might be a very useful alternative, especially in combination with *Hyperbolic Graph Convolutional Networks*[3]. We also optionally regularize sampled graphs with the different priors in [4]. To see whether our probabilistic approach also brings performance benefits compared to directly using the dense embedding similarity matrix, we tried disregarding the Bilevel formulation and directly optimize all components of our model on the training loss without any separation of inner or outer

optimization loop. We could not see noteworthy improvements from that which might also be due to insufficient exploration and hyper-parameter tuning. The most time during this project, however, was spent on hyper-parameter tuning and fiddling with sensitive parameters that made the more sensitive Bilevel optimization fail.

5 Implementation Details

We used *Pytorch*[22] as our automatic differentiation framework and for various data-loading and feature-processing steps we rely on the *Pytorch-Geometric*[8] and *Scikit-Learn*[23] libraries. *Sacred*[20] was used for experiment tracking together with *SEML*[33] to coordinate running the experiments on the lab's compute infrastructure.
In Pytorch it is not directly possible to calculate higher-order gradients that span over multiple weight updates as is necessary for meta-optimization. As soon as an optimizer in Pytorch updates a module's parameters, the computational graph is interrupted as modules are originally implemented to be stateful. Also, optimizers in Pytorch implement their operations on the gradient in a non-differentiable manner. We use the open-source package *Higher*[13] for differentiable implementations of optimizers and use the *Pytorch-Meta*[6] library to use a stateless abstraction of Pytorch modules. These simple modifications allow us to perform meta-optimization and could also be implemented without any extra dependencies in a few lines of code.

6 Experiments

As our main goal is to compare how our proposed method compares against LDS, we replicate the setup reported in their paper [11], if not otherwise stated. We test all methods on a variety of transductive classification problems.

6.1 Datasets

The primarily used datasets are the graph datasets *Cora* and *Citeseer*, presented in [30]. Both are citation networks where nodes represent papers and if one cites the other they are connected by an undirected edge. Node features come as tf-idf features of the contents of the papers. We also use two datasets from the *UCI Machine Learning Repository*[7], namely *Wine* and *Breast Cancer*. Other than Franceschi et al. we do not only evaluate on Cora and Citeseer with the train/ validation/ test splits used in [30], referred to as *Planetoid splits*, but also test all methods on random splits. [25] has shown that only using Planetoid splits is an unreliably way to compare models and the danger that also existing models are overfitted to these splits is a serious concern. For these two graph datasets, we also replicate the scenario where edge information is noisy by randomly removing 25%, 50% and 75% of all edges. It should be noted that in the official implementation of [11] after removing edges it can still be enforced that the graph is connected by re-adding some edges. We do not do this to simulate more realistic scenarios. Our results for these edge-deletion experiments might therefore differ. Dataset statistics are reported below in table 1. Except for Cora and Citeseer with the Planetoid splits, all experiments are repeated 10 times with different model initializations and random splits. As the model performance varies significantly between splits, the same 10 splits are used for different models. When running models on the Planetoid splits we use 10 different model initializations. As no prior graph information is available for the Wine and Breast Cancer datasets, we build a k-nearest-neighbor graph with $k \in \{10, 20\}$ using a distance metric $\in$ {minkowski, cosine} and report the best results for each.

6.2 Model Setup

For all models we performed grid-searches for the most important hyper-parameters and chose the others based on recommended values in the literature, mainly [11], [18] and [19]. Due to computational restrictions and since our proposed model needs longer training times than LDS, we, if not otherwise stated, did not perform a new grid-search for all dataset configurations but ran one for each model and always used the parameters that worked best on average.

GCN

Throughout all experiments, GCNs used for the classification part share the same hyper-parameters.

Name	Nodes	Edges	Features	Classes	Train/Val/Test
Citeseer	3,327	4,732	3,703	6	$120/500/1,000$
Cora	2,708	5,429	1,433	7	$140/500/1,000$
Wine	178	13	13	3	10/20/158
Cancer	569	30	30	2	10/20/539

Table 1: Dataset Statistics

They are based on [11], [19] and an initial grid-search. The GCN has 2 layers with a hidden layer with 16 dimensions. Gradient descent is performed using Adam[16] and a learning rate of 0.01. The parameters of the first layer are regularized with a weight-decay of 0.0005 and throughout the full network, dropout is used with a keep-probability of 50%. We use early-stopping with a patience of 20.

LDS

The hyper-parameters for our implementation of LDS are based on the values mentioned in the paper and the original implementation[10]. Due to computational limitations, we did not perform a grid-search for all parameter combinations for every dataset configuration but base our finally used parameters on a large initial grid-search that tries to replicate the one in [10]. The parameters of the graph distribution are also optimized using stochastic gradient-descent without momentum and we use an exponentially decaying learning rate of 0.1. We ran every experiment with a step-size of 10 and 20 for the outer optimization and each report the best results. Just like in the original paper we also found that increasing this value leads to better results but that improvement becomes neglectable for values larger than 10. To get the final output predictions we also draw 16 samples from the graph distribution and average the GCN's outputs. Just like for the GCN we also use early-stopping with a patience of 20.

Our Model

Although our model contains significantly less parameters, the computational graph's GPU memory requirement can amount to multiple gigabytes if the inner problem's training dynamics are unrolled over many epochs. Instead of going up to 20 as done in the LDS paper, we limit it to 15. For the initial grid search, we experimented with using both Adam and vanilla SGD without momentum and ultimately chose Adam. Each parameter configuration was re-run with another random seed 3 times and the results were averaged. The updates for the GAE parts use a learning rate of 0.01 whereas the learned bias and factor use a learning rate of 0.1. Like the original GAE model, we use a two-layer GCN. In the grid-search, we found that using dropout decreased performance but using a weight-decay of 0.0005 was beneficial. Out of multiple dimensionalities for the node embeddings we tried, higher values did not improve performance but actually significantly decreased it. Finally, we use a dimensionality of 16 for them. Unlike in [31] we use the cosine similarity instead of the pure dot-product in the decoder part as it performed slightly better. Adding the original graph to the predicted adjacency matrix as done in [31] did not bring the expected improvements but actually performed worse. However, we still do provide prior information about existing graph structure to the network by pre-training it as explained earlier. We also tried using no pre-training but this seemed to significantly decrease performance. To sparsify the sampled graph we use a k-nearest-neighbor graph. We tried multiple values between 5 and even went up to 200 in our experiments but ultimately try $k \in \{10, 20\}$ for each dataset configuration and choose the best performing one. Using a threshold for the cosine similarity instead did not lead to competitive results. We also tried multiple combinations of the graph-priors presented in [4], with either $\{0, 0.01, 0.001\}$ for each but it consistently decreased performance. We, therefore, do not use any additional prior for the predicted graphs. We also tried to optimize our models without any Bilevel-formulation just like in [31, 4] but always achieved worse results than using the training regime of LDS. To summarize: our final graph distribution is a GAE that uses the two methods explained in 4.1 to thin out the graph before feeding it to the classification GCN. We train it just

	Cora	Citeseer	Wine	Breast Cancer
GCN	81.1 ± 1.0	68.7 ± 2.2	95.5 ± 1.0	93.4 ± 1.8
LDS	81.2 ± 1.5	$\mathbf{72.5 \pm 1.7}$	$\mathbf{95.6 \pm 1.4}$	$\mathbf{93.7 \pm 1.6}$
Ours	$\mathbf{82.5 \pm 1.3}$	70.6 ± 1.3	95.5 ± 1.9	93.2 ± 2.0

Table 2: Model Comparison.
Test accuracies are reported for the same 10 random splits. For Wine and Breast Cancer all methods start with a k-nearest-neighbor graph. Best model for each dataset is shown with bold font.

like LDS but optimize over fewer inner optimization steps to still be able to train it on a GPU with limited memory.

6.3 Results

Table 2 shows each methods results on all datasets. For Cora and Citeseer we report results using random splits. The full available graph is used for these two and the mentioned k-nearest-neighbor graph for the UCI datasets. We see that, except for Cora, LDS performs better than a normal GCN and our proposed method, indicating that the ability to directly learn individual edges still has an advantage over calculating them via learned node embeddings. However, unlike the results reported in [11], the advantage of using LDS for Wine and Breast Cancer is far lower, deserving further investigation. Our method seems to not be useful for these two datasets. But based on the results on Cora and Citeseer we see that our method seems to work as well as we initially hoped at least on datasets where initial, high-quality graph information is available. It was already mentioned that the model performance strongly depends on the used splits. We, therefore, compare how a GCN, LDS and our method compare when using the original Planetoid splits in table 3. There we see that

	Cora	Citeseer
GCN	81.2 ± 0.4	70.8 ± 0.5
LDS	$\mathbf{84.2 \pm 0.5}$	$\mathbf{74.0 \pm 0.5}$
Ours	82.4 ± 1.1	71.8 ± 1.1

Table 3: Test Accuracies on Planetoid Splits.
Results are averaged for 10 randomly initialized models.

the overall performance of all models increases, especially for LDS, by multiple percent in accuracy, when using Planetoid splits. This finding is on-par with what is reported in [25] and [31]. LDS apparently is overfitted to the Planetoid splits as are many other methods. This table again shows that we perform better than a normal GCN but still do not completely reach the performance of LDS. The authors of LDS report that their method is more robust to deleted edges in the original graph and we, therefore, perform the same experiment as them. In figure 1 it is shown how the accuracy on the test set declines when an increasing number of random edges in the graphs are deleted. Note that unlike done in [11] we deliberately do not re-add some of these edges to enforce that the graph is connected. As we want to see in this experiment how real-world noise in the graph might affect the models, we consider our setup more useful. Similar to the findings in [11] we can see that a GCN is severely affected, especially when about 75% of edges are removed the accuracy drops by up to 10%. Similar to what we saw before, as we perform this experiment using random instead of Planetoid splits, the improvements of LDS are not as significant as in [11] but still notable, especially for Citeseer. Our method performs similar to LDS, for Cora slightly better and Citeseer slightly worse, however always better than a normal GCN. It, therefore, indicates that our method is especially effective when many edges in a given graph are missing. One further result we are interested in is whether the edges learned by our model bear any meaning or can be interpreted in any useful way. Figure 2 shows how the probabilities that nodes from the same class vs from a different class evolve in the course of the training process. We see that our model is far more likely to predict an edge between two nodes if they come from the same class. For nodes of different classes, the mean edge probability is around 5%.

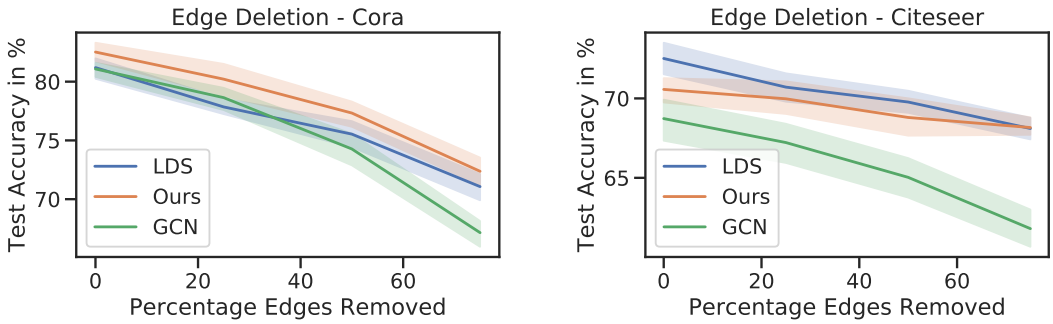

Figure 1: Edge Deletion Scenario for Cora and Citeseer.
We show the decline of each model's test set accuracy for an increasing percentages of random edges that are removed from the graph. For each percentage we re-run each model with 10 random dataset splits and parameter initializations.

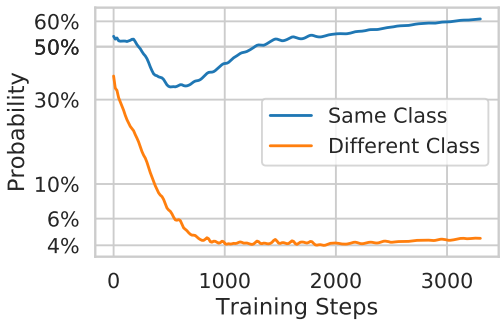

Figure 2: Edge Probability Analysis for a randomly picked training run on Cora. Edges between nodes from different classes are on-average attributed much lower probabilities than edges between nodes from the same class.

Finally, we would like to show how our method compares to the two methods presented before, namely GRCN3.2 and DIAL-GNN3.3. As these two methods were published while we worked on ours and as for neither a public implementation exists at the time of writing, we could not compare them in detail. It has to be mentioned though that both GRCN as well as DIAL-GNN, according to their respective experiments, outperform LDS and our method on all of our benchmark datasets without exception. The specific results reported in these papers are listed in table 4. Especially since GRCN is very similar to our method or variations we tried, we suspect that their simpler, non-probabilistic approach and simpler optimization is mainly responsible for these results. However, this deserves further analysis and proper ablation studies are vital.

7 Conclusion

Based on the work of Franceschi et al. we explain why basic GNNs like the GCN struggle when no graph is available for a problem or when an existing graph is noisy or incomplete. It is demonstrated why their method LDS does not scale to large graphs and we present a modification of their approach that scales better to large graphs as the number of its parameters does not depend on the size of the graph. Multiple variants of our method are proposed and special emphasis is put on how a predicted graph can be sparsified for better performance. In our experiments on different datasets, we show that our method does not generally perform better than LDS but that it still performs better than a simple GCN on a variety of tasks. Albeit providing better scalability we were still not able to run our method on large datasets due to the computationally expensive optimization procedure we copy from LDS. In our result analysis, we also show how two similar approaches that were recently published that are inherently easier to optimize outperform both our method and LDS. We think that future could should be to compare these methods with ours and perform proper ablation studies to see which components of the different models and optimization methods

	Cora		Citeseer	
	Planetoid	Random	Planetoid	Random
Ours	82.4 ± 1.1	82.5 ± 1.3	71.8 ± 1.1	70.6 ± 1.3
LDS	84.2 ± 0.5	81.2 ± 1.5	74.0 ± 0.5	72.5 ± 1.7
[31]	N/A	$\mathbf{83.9 \pm 1.7}$	N/A	$\mathbf{72.6 \pm 1.3}$
[4]	$\mathbf{84.5 \pm 0.3}$	N/A	$\mathbf{74.1 \pm 0.2}$	N/A

Table 4: Comparison with Similar Methods
Test Accuracies are taken from the respective papers and our experiments. Results are shown separately for different splits.

prove to be most useful.

References

[1] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv*, 2018.

[2] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[3] Ines Chami, Zhitao Ying, Christopher Ré, and Jure Leskovec. Hyperbolic graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 4869–4880, 2019.

[4] Yu Chen, Lingfei Wu, and Mohammed J Zaki. Deep iterative and adaptive learning for graph neural networks. *arXiv preprint arXiv:1912.07832*, 2019.

[5] Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of operations research*, 153(1):235–256, 2007.

[6] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, and Yoshua Bengio. Torchmeta: A Meta-Learning library for PyTorch, 2019. Available at: https://github.com/tristandeleu/pytorch-meta.

[7] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[8] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[9] Luca Franceschi, Paolo Frasconi, Saverio Salzo, Riccardo Grazzi, and Massimiliano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1568–1577, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[10] Luca Franceschi, Mathias Niepert, Massimiliano Pontil, and Xiao He. LDS-GNN. `https://github.com/lucfra/LDS-GNN`, 2019.

[11] Luca Franceschi, Mathias Niepert, Massimiliano Pontil, and Xiao He. Learning discrete structures for graph neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1972–1982, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[12] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

[13] Edward Grefenstette, Brandon Amos, Denis Yarats, Phu Mon Htut, Artem Molchanov, Franziska Meier, Douwe Kiela, Kyunghyun Cho, and Soumith Chintala. Generalized inner loop meta-learning. *arXiv preprint arXiv:1910.01727*, 2019.

[14] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[15] Bo Jiang, Ziyan Zhang, Doudou Lin, Jin Tang, and Bin Luo. Semi-supervised learning with graph learning-convolutional networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11305–11312, 2019.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[18] Thomas N. Kipf and Max Welling. Variational graph auto-encoders. *CoRR*, abs/1611.07308, 2016.

[19] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[20] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49 – 56, 2017.

[21] Maximillian Nickel and Douwe Kiela. Poincaré embeddings for learning hierarchical representations. In *Advances in neural information processing systems*, pages 6338–6347, 2017.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[23] 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*, 12:2825–2830, 2011.

[24] Darwin Saire Pilco and Adín Ramírez Rivera. Graph learning network: A structure learning algorithm. *arXiv preprint arXiv:1905.12665*, 2019.

[25] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.

[26] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[27] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[28] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[29] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural

networks. *arXiv preprint arXiv:1901.00596*, 2019.

[30] Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pages 40–48, 2016.

[31] Donghan Yu, Ruohong Zhang, Zhengbao Jiang, Yuexin Wu, and Yiming Yang. Graph-revised convolutional network. *arXiv preprint arXiv:1911.07123*, 2019.

[32] Daniel Zügner and Stephan Günnemann. Certifiable robustness and robust training for graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 246–256, 2019.

[33] Daniel Zügner and Johannes Klicpera. Seml: Slurm experiment management library. `https://github.com/TUM-DAML/seml`, 2019.