Revisiting Over-smoothing in Deep GCNs

Chaoqi Yang, Ruijie Wang, Shuochao Yao, Shengzhong Liu, Tarek Abdelzaher
University of Illinois, Urbana-Champaign, IL 61801, USA
{chaoqiy2,ruijiew2,syao9,sl29,zaher}@illinois.edu

Abstract

Oversmoothing has been assumed to be the major cause of performance drop in deep graph convolutional networks (GCNs). In this paper, we propose a new view that *deep GCNs can actually learn to anti-oversmooth during training*. This work interprets a standard GCN architecture as layerwise integration of a Multi-layer Perceptron (MLP) and graph regularization. We analyze and conclude that before training, the final representation of a deep GCN does over-smooth, however, it learns anti-oversmoothing during training. Based on the conclusion, the paper further designs a cheap but effective trick to improve GCN training. We verify our conclusions and evaluate the trick on three citation networks and further provide insights on neighborhood aggregation in GCNs.

1 Introduction

Graph neural networks (GNNs) are widely used in modeling real-world graphs, like protein networks (Ying et al. 2018), social networks (Veličković et al. 2018), and co-author networks (Kipf and Welling 2016). One could also construct similarity graphs by linking data points that are close in the feature space even when there is no explicit graph structure. There have been several successful GNN architectures: ChebyshevNet (Defferrard, Bresson, and Vandergheynst 2016), GCN (Kipf and Welling 2016), MPNN (Gilmer et al. 2017), SGC (Wu et al. 2019), GAT (Veličković et al. 2017), GraphSAGE (Hamilton, Ying, and Leskovec 2017) and other subsequent variants tailored for practical applications (Gilmer et al. 2017; Liu, Nickel, and Kiela 2019; Klicpera, Bojchevski, and Günnemann 2018).

Recently, researchers started to explore the fundamentals of GCNs (Kipf and Welling 2016), such as expressive power (Xu et al. 2018a; Oono and Suzuki 2019; Loukas 2019; Dehmamy, Barabási, and Yu 2019), and analyze their capacity and limitations. One of the frequently mentioned problem during GCN training is *oversmoothing* (Li et al. 2019; Zhou et al. 2018). In deep graph convolution based architectures, *over-smoothing* means that after multi-layer graph convolution, the effect of Laplacian smoothing makes node representations more and more similar, which eventually become indistinguishable. This issue was first mentioned in (Li, Han, and Wu 2018) and has been widely discussed since then, such as in DenseGCN (Li et al. 2019), DropEdge (Rong et al. 2019), PairNorm (Zhao and Akoglu 2019) and BVAT (Deng, Dong, and Zhu 2019). Recently, (Chen et al. 2019) puts forth that reasonable smoothing could makes GCN work, and oversmoothing would cause the bad performance.

In this work, we propose a new understanding that *though deep GCNs does lead to oversmoothing with initial parameters (before training), it can learn anti-oversmoothing during training.* This paper starts from the perspective of graph-based regularization model (two loss functions $\mathcal{L}_0 + \gamma\mathcal{L}_{reg}$ as supervision) (Belkin and Niyogi 2003; Yang, Cohen, and Salakhutdinov 2016), where $\mathcal{L}_0$ is the empirical loss and $\mathcal{L}_{reg}$ is a graph regularizer, which encodes smoothness over the connected node pairs.

We transform the MLP-based graph regularization model by two steps (and minimizing two loss functions respectively in each step), which gives the GCN model. STEP1 encodes the graph regularizer $\mathcal{L}_{reg}$ implicitly into the layerwise propagation of MLP, resulting in the GCN architecture (before renormalization); STEP2 optimizes the parameters by conducting standard back-propagation algorithm on $\mathcal{L}_0$, under the new architecture. Therefore, GCN could be expressed conceptually as a two-step minimization:

$$\underbrace{\text{STEP1 (layerwise): } \min_{\{X^{(l)}\}} \mathcal{L}_{reg}\left(\{X^{(l)}\} \mid \{W^{(l)}\}\right)}_{why\ GCN\ architecture?}$$

$$\text{and} \quad \underbrace{\text{STEP2: } \min_{\{W^{(l)}\}} \mathcal{L}_0\left(\{W^{(l)}\}\right)}_{train\ the\ architecture.}.$$

When viewed as a graph regularization problem, GCN essentially encodes $\mathcal{L}_{reg}$ in the forward propagation rule (architecture) and then optimizes the parameters under the supervision of $\mathcal{L}_0$.

From this reformulation, we can clearly know that before training (after STEP1), deep GCNs do suffer from oversmoothing. Because the effect of deep GCN architecture will naturally minimize $\mathcal{L}_{reg}$, which gradually makes all the node representations proportional to the largest eigenvector of the Laplacian (we show it in Section 3). However, during STEP2/training, GCNs will learn to prevent oversmoothing because (i) the oversmoothing situation is conditioned on $\{W^{(l)}\}$; (ii) the explicit goal of STEP2/training is to find optimal $\{W^{(l)}\}$, so as to minimize empirical loss $\mathcal{L}_0$ and (iii)

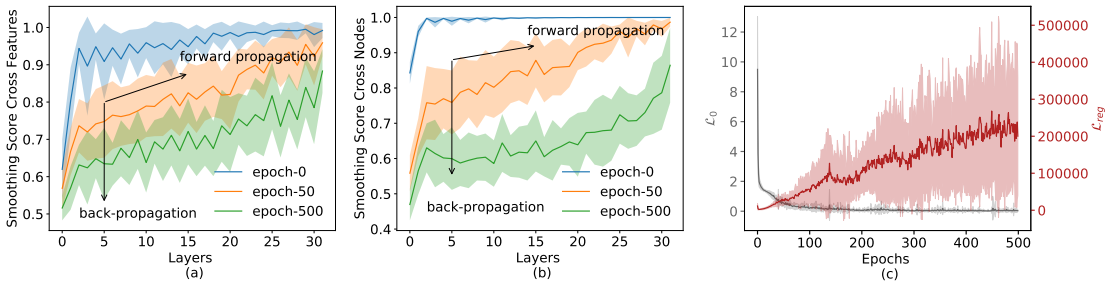

Figure 1: Anti-oversmoothing Demo on *Karate* Dataset. The *Karate* graph has 34 vertices of 4 classes (the same labeling strategy as (Kipf and Welling 2016; Perozzi, Al-Rfou, and Skiena 2014)) and 78 edges. Each class has two labeled samples. Initial feature vectors are uniformly randomized. We apply 32-layer GCN with 16 hidden units on loss $\mathcal{L}_0$ (for 500 epochs). For the output of each layer, we compute feature-wise smoothing score in Fig. 1.(a) and node-wise smoothing score in Fig. 1.(b). They are calculated by vector cosine similarity (details are given in Appendix G). Fig. 1.(c) shows two loss functions $\mathcal{L}_0$ and $\mathcal{L}_{reg}$ for each training epoch. Within one epoch, we observe that nodes/features are becoming more and more indistinguishable through forward propagation. For any layer, we observe that the smoothness does disappear gradually during back-propagation/training.

as long as the oversmoothing exists, feature representations will be indistinguishable, so upon minimizing $\mathcal{L}_0$, the model *must* learn to make features separable, which naturally means anti-oversmoothing (see the demo in Figure 1).

Based on the reformulation, we further propose a *mean-subtraction* trick. We show that applying mean-subtraction layerwise is equivalent to approximate the Fiedler vector (the second smallest eigenvector of the Laplacian), which set an initial graph partition and speeds up the training.

In the experiment, we empirically verify that it is not over-smoothing that leads to a performance drop in training deep GCNs. Instead, we conjecture that *overfitting* might be the major contributing factor based on experimental evidence. The experiments also demonstrate the efficacy of the proposed mean-subtraction trick and provide more insights on neighborhood aggregation in GCNs.

2 Graph Transductive Learning

Graph representation learning aims at embedding the nodes into low-dimensional vectors, while simultaneously preserving both *graph topology structure* and *node feature information*. Afterwards, one then apply subsequent downstream learning algorithms for tasks, like node classification.

Given a graph $G = (V, E)$, let $V = \{v_1, v_2, \cdots, v_n\}$ be the set of nodes, and let Y be a set of m possible classes. Assume that each node v_j is associated with a class label $y_j \in Y$. A graph could be represented by an *adjacency matrix* A with $A_{ij} = 1$ when two nodes are connected $(v_i, v_j) \in E$. The *degree matrix* $D = diag(d_1, d_2, \ldots, d_n)$ is diagonal where $d_i = \sum_j A_{ij}$. Let $X = \{x_1, x_2, \ldots, x_n\}$ denote the feature vectors for each node. Given a labelled set $T \subset V$, the goal of transductive learning on a graph is to transductively predict labels for the remaining unknown nodes $V \setminus T$. A well-studied solution category is to include graph regularizers (Belkin and Niyogi 2003; Tenenbaum, De Silva, and Langford 2000; Zhu and Ghahramani 2002; Weston et al. 2012) into the classification algorithm. Graph convolution based models (Kipf and Welling 2016; Defferrard, Bresson, and Vandergheynst 2016; Veličković et al. 2017; Hamilton, Ying, and Leskovec 2017) are also powerful learning approaches in this space.

2.1 Graph-based Regularization

There is a rather general class of embedding algorithms that include graph regularizers. They could be described as: finding a mapping $f(\cdot)$, i.e., MLP, by minimizing the following two-fold loss:

$$\mathcal{L} = \mathcal{L}_0\left(f(X)\right) + \gamma \mathcal{L}_{reg}\left(f(X)\right), \tag{1}$$

where $f(X) = [f(x_i)]_{i=1}^n$ is the low-dimensional representation of nodes, and γ denotes the weight. The first term is the empirical risk on the labelled set T. The second term is a graph regularizer over the connected pairs, so as to make sure that a trivial solution is not reached.

The measurements on graphs are usually invariant to node permutations. A canonical way is to use Dirichlet energy (Belkin and Niyogi 2002) for the graph-base regularization,

$$\begin{aligned}\mathcal{L}_{reg} &= \frac{1}{2}\sum_{i,j} A_{ij} \left\| \frac{f(x_i)}{\sqrt{d_i}} - \frac{f(x_j)}{\sqrt{d_j}} \right\|^2 \\ &= \frac{1}{2}\,\mathrm{Tr}\left(f(X)^\top \Delta f(X)\right),\end{aligned} \tag{2}$$

where $\Delta = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ is the *normalized Laplacian operator*, which induces a semi-norm on $f(\cdot)$, penalizing the changes between adjacent vertices. Same normalized formulation could be found in (Chen, Wu, and Zaki 2019; Ando and Zhang 2007; Smola and Kondor 2003; Shaham et al. 2018; Belkin and Niyogi 2003), and some related literature also use the unnormalized version (Zhu and Ghahramani 2002; Von Luxburg 2007).

2.2 Graph Convolutional Network

GCNs are derived from graph signal processing (Sandryhaila and Moura 2013; Chen et al. 2015; Duvenaud et al. 2015).

On the spectral domain, the operator Δ is a real-valued symmetric semidefinite matrix and the graph convolution is parameterized by a learnable filter g_θ on its eigenvalue matrix. Kipf et al. (Kipf and Welling 2016) made assumptions of the largest eigenvalue (i.e., $\lambda_{max} = 2$) and simplified it with two-order Chebyshev expansion,

$$g_\theta \star x \approx \theta_0 x + \theta_1(\Delta - I)x \approx \theta(I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})x. \quad (3)$$

A multi-layer graph convolutional network (GCN) is formulated as the following layerwise propagation rule ($\sigma(\cdot)$ is an activation function, e.g., ReLU):

$$X^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}(I + A)\tilde{D}^{-\frac{1}{2}}X^{(l)}W^{(l)}\right) \quad (4)$$

where $\tilde{D}^{-\frac{1}{2}}(I + A)\tilde{D}^{-\frac{1}{2}} \leftarrow I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the renormalization trick, $X^{(l)}$ and $W^{(l)}$ are the layerwise feature and parameter matrices, respectively.

3 GCN as Layerwise Integration of Graph Regularizer and MLP

These two broad graph representation algorithms are closely related. In this section, we reformulate GCN (in Sec. 2.2) from the MLP-based graph regularization algorithm (in Sec. 2.1). Essentially, we combine the MLP architecture and the gradient descent rule of minimizing $\mathcal{L}_{reg}$. We show that the resulting architecture is identical to the GCN before re-normalization. Let us first discuss a gradient descent algorithm to minimize $\mathcal{L}_{reg}$.

3.1 Gradient Descent for Minimizing $\mathcal{L}_{reg}$

Given the Laplacian operator $\Delta \in \mathbb{R}^{n\times n}$, we consider to minimize the graph regularizer $\mathcal{L}_{reg} = \frac{1}{2}\,\mathrm{Tr}(X^\top \Delta X)$ on feature domain $X \in \mathbb{R}^{n\times d}$, where d is the input dimension. To prevent the trivial solution $X = \mathbf{0} \in \mathbb{R}^{n\times d}$, we consider the energy constraint on X, i.e., $\|X\|_F^2 = c_1 \in \mathbb{R}^+$. The trace optimization problem is:

$$\min \ \frac{1}{2}\,\mathrm{Tr}(X^\top \Delta X),\ \textit{subject to const.}\ \|X\|_F^2, \quad (5)$$

where $\|X\|_F^2$ denotes the Forbenius-norm of X. To solve this, We equivalently transform the optimization problem into the *Reyleigh Quotient* form $R(X)$, which is,

$$\min \ R(X) = \frac{\frac{1}{2}\,\mathrm{Tr}(X^\top \Delta X)}{\|X\|_F^2} = \frac{\frac{1}{2}\,\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)},$$
$$\textit{subject to constant}\ \|X\|_F^2. \quad (6)$$

It is obvious that $R(X)$ is scaling invariant on X, i.e., $\forall\, c_2 \neq 0 \in \mathbb{R}$, $R(X) = R(c_2 \cdot X)$.

One-step Improvement. Given an initial guess, X, one-step of trace optimization aims at finding a better guess X_{better}, which satisfies $R(X_{better}) \leq R(X)$ and $\|X_{better}\|_F^2 = c_1$. Our strategy is first viewing the problem as unconstrained optimization on $R(X)$ and update the guess X to the intermediate value X_{mid}, such that $R(X_{mid}) \leq R(X)$, by gradient descent. Then we rescale X_{mid} to reach the improved guess X_{better}, which meets the norm constraint.

Given the initial guess X, we apply gradient descent with learning rate $\eta = \frac{\mathrm{Tr}(X^\top X)}{2-\frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}}$ and reach an intermediate solution X_{mid} in the unconstrained space:

$$\nabla_X = \frac{\partial R(X)}{\partial X} = \frac{1}{2}\frac{\partial \frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}}{\partial X} = \frac{\left(\Delta - I\frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}\right)X}{\mathrm{Tr}(X^\top X)}, \quad (7)$$

$$X_{mid} = X - \eta\nabla_X = \frac{(2-\Delta)X}{2 - \frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}} = \frac{(I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})}{2 - \frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}}X. \quad (8)$$

Immediately, we get $R(X_{mid}) \leq R(X)$ (proofs in Appendix B). Then, we rescale X_{mid} by a constant $c_3 \in \mathbb{R}^+$, i.e., $X_{better} = c_3 \cdot X_{mid}$, so as to meet the norm constraint, i.e., $\|X_{better}\|_F^2 = c_1$.

Discussion of the Form. In sum, we reach a better guess X_{better}, which satisfies $R(X_{better}) = R(X_{mid}) \leq R(X)$ and follows this form,

$$X_{better} = c_3 \cdot X_{mid} \propto (I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})X. \quad (9)$$

Note that, the eigenvectors of $(I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})$ and Δ are the same. The operator $(I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})$ in Equation (9) is similar to an one-step *Laplacian* smoothing, which explains why it will give a better guess in terms of minimizing $\mathcal{L}_{reg}$. However, it causes the issue of *oversmoothing* after sufficient number of layers. We will discuss in depth in Sec. 4. It is also interesting that when combining with MLP architecture, the only magic in Equation (9) is the operator $(I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})$, since the scalar will be absorbed into the layerwise parameter matrix of MLP. Let us discuss this below.

3.2 Layerwise Propagation and Optimization

We introduce the solution from Sec. 3.1 into the layerwise propagation of MLP. Given the node set V, features $X = \{x_1, x_2, \ldots, x_n\}$ and a labelled set $T \subset V$, a label mapping, $f_{\{W^{(l)}\}} : X \mapsto Y$, is usually a deep neural network, which could be tailored according to the practical applications. In this scenario, we consider a simple multi-layer perceptron (MLP). The forward propagation rule of an standard MLP is given by,

$$X^{(l+1)} = \sigma(X^{(l)}W^{(l)}),\ \ l = 1, \ldots, L \quad (10)$$

where $X^{(0)}$ is the feature matrix, $W^{(l)}$ and $X^{(l)}$ are layerwise parameters and inputs.

STEP1: minimizing $\mathcal{L}_{reg}$ in Forward Propagation. To ensure that the output of MLP could lead to a smaller $\mathcal{L}_{reg}$, an intuitive way is to apply the gradient descent step between the layerwise propagation, so that the smoothing effect will be accumulated layer-by-layer towards the final representation. Let us consider the output of the $(l-1)$-th layer, i.e., $X^{(l)}$. We know from Sec. 3.1 that through one-step gradient descent, it transforms into:

$$X^{(l)}_{better} \propto (I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})X^{(l)}. \quad (11)$$

We plug this new value into Eqn. (10) and immediately reach the same convolutional propagation rule, $f^{(l+1)} : X^{(l)} \mapsto X^{(l+1)}$, as Kipf et al. (Kipf and Welling 2016) (before applying the renormalization trick),

$$\begin{aligned} X^{(l+1)} &= \sigma(X^{(l)}_{better} W^{(l)}) \\ &= \sigma\left((I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}) X^{(l)} W^{(l)}_{new}\right), \end{aligned} \tag{12}$$

where the constant scalar in Eqn. (11) is absorbed into parameter matrix $W^{(l)}$, resulting in $W^{(l)}_{new}$ (we still use the notation $W^{(l)}$ below if there is no ambiguity). Therefore, a GCN forward propagation is essentially applying STEP1 layerwise in the forward propagation of an MLP, which is a composition of mappings $f = f^{(L)} \circ \cdots \circ f^{(1)}$ on initial feature $X^{(0)}$. In essence, the GCN structure implicitly contains the goal of minimizing graph regularizer.

STEP2: minimizing $\mathcal{L}_0$ in Back Propagation. From the above, we have transformed the graph regularization $\mathcal{L}_{reg}$ as a layerwise convolution operator. Then the empirical loss $\mathcal{L}_0$ will be the only supervision. In the model training process, the standard back-propagation algorithm is used on $\mathcal{L}_0$.

3.3 GCN: combining STEP1 and STEP2

In sum, the GCN model can be interpreted from a graph regularization view. STEP1 encodes the graph regularizer implicitly into an MLP propagation, which explains "why GCN architecture?" In STEP2, under that architecture, the optimal $\{W^{(l)}\}$ is learned and a low-dimension $f(X)$ is reached with respect to $\mathcal{L}_0$ explicitly and $\mathcal{L}_{reg}$ implicitly, after standard loss back-propagation. When viewed as a graph regularization problem, GCN could be expressed conceptually as a two-step optimization,

$$\underbrace{\text{STEP1 (layerwise): } \min_{\{X^{(l)}\}} \mathcal{L}_{reg}\left(\{X^{(l)}\} \mid \{W^{(l)}\}\right)}_{why\ GCN\ architecture?}$$

$$\text{and} \quad \underbrace{\text{STEP2: } \min_{\{W^{(l)}\}} \mathcal{L}_0\left(\{W^{(l)}\}\right)}_{train\ the\ architecture.}. \tag{13}$$

In this section, the learning rate $\eta = \frac{\text{Tr}(X^\top X)}{2 - \frac{\text{Tr}(X^\top \Delta X)}{\text{Tr}(X^\top X)}}$ is specially chosen, and it satisfies $\eta \in (0, \infty)$ since $X^\top X$ is semi-definite and $\frac{\text{Tr}(X^\top \Delta X)}{\text{Tr}(X^\top X)}$ is smaller than the largest eigenvalue of Δ, which is smaller than 2. In the experiment section, we reveal that η is related to the weight of neighborhood aggregation. We further test different η and provide more insights on how to set the aggregation weights in Sec. 5.3 experiments. In the following sections, we use A_{sym} to denote the re-normalized convolutional operator $\tilde{D}^{-\frac{1}{2}}(I + A)\tilde{D}^{-\frac{1}{2}}$ and use A_{rw} for the random walk form $\tilde{D}^{-1}(I + A)$.

4 Analysis and Improvement

The recent successes in applying GNNs are largely limited to *shallow* architectures (e.g., 2-4 layers). Model performance decreases when adding more intermediate layers. Summarized in (Zhao and Akoglu 2019), there are three possible contributing factors: (i) overfitting due to increasing number of parameters; (ii) gradient vanishing/exploding; (iii) over-smoothing due to Laplacian smoothing. The first two points are common in all deep architectures. The issue of over-smoothing is therefore our focus in the section. We analyze the behavior of deep GCNs in terms of the training process and conclude that *the training process of GCN starts from the oversmoothing situation, and deep GCNs can learn to anti-oversmooth* (in the experiment, we show that *overfitting* might be the major factor for performance drop). Based on the analysis, we further propose a cheap but effective trick to speed up deep GCNs training.

4.1 Conditional Over-smoothing Before Training

Oversmoothing means that node representations become more and more similar and finally go indistinguishable after multi-layer graph convolution. Previous literature (Li, Han, and Wu 2018; Xu et al. 2018b; Rong et al. 2019; Li et al. 2019; NT and Maehara 2019; Zhao and Akoglu 2019) already discussed that due to the Laplacian smoothing effect, deep GCN architectures lead to oversmoothing. The primary reason is summarized in Theorem 1 (see proofs in Appendix D). (Oono and Suzuki 2019) further provides a similar result while considering the ReLU activation function during the analysis, under the assumption that the singular values of each parameter matrix are bounded by 1.

Theorem 1. *Given any random signal $x \in \mathbb{R}^n$ and a (symmetric) adjacency matrix $A \in \mathbb{R}^{n \times n}$, the following property $\lim_{k \to \infty} A^k x \propto u_1$ holds almost everywhere on x, where A has non-negative eigenvalues and u_1 is the eigenvector associated with the largest eigenvalue of A.*

Specifically, for two widely used convolution operators A_{sym} and A_{rw}, they have the same dominant eigenvalue $\lambda_1 = 1$ with eigenvectors $\tilde{D}^{\frac{1}{2}}\mathbf{1}$ and $\mathbf{1}$, respectively. Before training, it is intuitive that if the depth L goes to infinity, then the output channel of GCN will become proportional to $\tilde{D}^{\frac{1}{2}}\mathbf{1}$ or $\mathbf{1}$.

Luckily, this situation is conditioned on the parameter matrices. During training, the learned parameters can substantially reverse the smoothing effect through the non-linear functions (in Appendix E, we analyze SGC (Wu et al. 2019) similarly, which is a linear version of GCN. We prove that without the activation function, the oversmoothing of SGC will be independent of the parameters and thus SGC cannot learn to anti-oversmooth). During training, GCN will learn to address the oversmoothing issue gradually.

4.2 Anti-oversmoothing During Training

We show in Sec. 3 that GCN could be expressed as a two-step optimization, where the STEP1 minimizes $\mathcal{L}_{reg}$ conditioned on $\{W^{(l)}\}$, and the STEP2 finds optimal $\{W^{(l)}\}$ and minimizes $\mathcal{L}_0$. In this section, we shall analyze why STEP2 will learn anti-oversmoothing naturally.

Analysis. Before training the GCN architecture, we know from Theorem 1 that the output representation of each feature channel will be proportional to the largest eigenvector of the convolution operator. In that case, the feature representations are indistinguishable and lead to a small graph regularization loss $\mathcal{L}_{reg}$ but a large supervised loss $\mathcal{L}_0$ (like Figure 1.(c)). The training process of GCN is to re-balance the trade-off between $\mathcal{L}_{reg}$ and $\mathcal{L}_0$ from the initial oversmoothing situation.

$\mathcal{L}_{reg}$ encodes the smoothness over the connected node pairs, which favors the solution, where the connected nodes share similar representations ("similar" means the scale of each feature channel is approximately proportional to the square root of its degree, refer to Equation (2), and oversmoothing is an extreme case). $\mathcal{L}_0$ calculates the error based on labels. If the labels are well-aligned with node degree information, then $\mathcal{L}_{reg}$ and $\mathcal{L}_0$ will aim to learn similar representations and the feature matrices will not change a lot during training. However, in real practice, the labels usually contain other semantic information, so that the goals of two loss functions, i.e., $\mathcal{L}_{reg}$ and $\mathcal{L}_0$, are not always aligned. Since the explicit supervision of STEP2/training is $\mathcal{L}_0$, the training process is actually a step-by-step shift from the oversmoothing region (where node representations are proportional to the square root of the degrees) towards the optimal region (where node representations are partially similar, separable and well-aligned with the labeling semantics), which naturally means anti-oversmoothing.

4.3 Improve Deep GCN Training

As revealed above, the learning of deep GCN begins with the oversmoothing situation, which makes the training slow. This issue has not been explored extensively in the literature (Chen, Ma, and Xiao 2018; Chiang et al. 2019). In this work, we propose a cheap but effective trick to ensure a better beginning point and accelerate GCN training.

Motivation. We propose *mean-subtraction*, i.e., reducing the mean value from each feature channel of each hidden layer. Our motivation primarily stems from Theorem 1, where the *Power Iteration* of convolution operator is to approximate the largest eigenvector, which causes an oversmoothing start. After applying *mean-subtraction*, the revised *Power Iteration* will lead to the Fiedler vector (the second smallest eigenvector), which provides a coarse graph partition result and makes the training faster. PairNorm (Zhao and Akoglu 2019) also includes a *mean-subtraction* step, however, the authors did not state extensively in their paper. Our paper instead analyzes the mechanism.

Mean-subtraction. We start with operator A_{rw} and its largest eigenvector $u_1 = \mathbf{1} \in \mathbb{R}^n$. For any output feature channel k of the l-th layer, i.e., $X_k^{(l)} \in \mathbb{R}^n$, the *mean-subtraction* gives,

$$X_k^{(l)} \leftarrow X_k^{(l)} - \bar{X}_k^{(l)} = X_k^{(l)} - \frac{\mathbf{1}\mathbf{1}^\top X_k^{(l)}}{n} = X_k^{(l)} - \langle X_k^{(l)}, \bar{u_1} \rangle \cdot \bar{u_1} \quad (14)$$

where $\bar{u_1} = \frac{u_1}{\|u_1\|}$. Eqn. (14) essentially reduces the components aligned with $\{u_1\}$-space. This is exactly one-step approximation of the Fiedler vector by *Power Iteration*. Fiedler vector is widely used to partition a graph (Chung and Graham 1997) in spectral graph theory, and it separates nodes initially. In essence, the vanilla GCN models train from the oversmoothing stage. With *mean-subtraction* trick, the revised GCNs will consequently train on a coarse graph partition result (demo in Fig. 1), which is much faster. For the symmetric operator A_{sys}, the formulation will be adjusted by a factor, $\tilde{D}^{\frac{1}{2}}$ (refer to derivation in Appendix F). We show the power of *mean-subtraction* in Sec. 5.2.

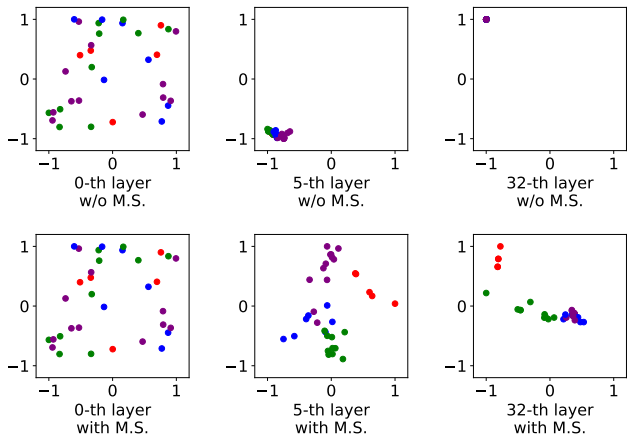

Figure 2: Untrained GCNs on *Karate* with or w/o Mean-subtraction (M.S.). It is impressive that with mean-subtraction, nodes are almost well-separated with initial GCN parameters and uniformly randomized feature vectors. The experiment setting is as follows: we run an untrained 32-layer GCNs with 16 hidden units on *Karate* with normalized random-walk adjacency operator A_{rw}. We visualize the first two feature channels f_1, f_2 of each node after scaling the dimension by the largest absolute value (i.e., $f = f$ / max(abs(f))) for the k-th layer ($k = 0, 5, 32$). $k = 0$ is for initial feature. From ground truth, each color indicates a class and we manually add them to help with the visualization.

Table 1: Overview of Citation Network Statistics

Dataset	#Nodes	#Edges	#Features	#Class	Label rate
Cora	2,708	5,429	1,433	7	0.052
Citeseer	3,327	4,732	4,732	6	0.036
Pubmed	19,717	44,338	500	3	0.003

5 Experiments

In this section, we present experimental evidence on *Cora*, *Citeseer*, *Pubmed* to answer the following questions: (i) what is the real cause of performance drop in deep GCNs and why? (ii) How to improve (accelerate and stabilize) the training of a generic deep GCN model? (iii) Does the learning rate η (defined in Sec. 3) matter? How to choose the weights of neighborhood aggregation?

Experiment Setup. The experiments are basically on semi-supervised node classification tasks. We use *ReLU* as the

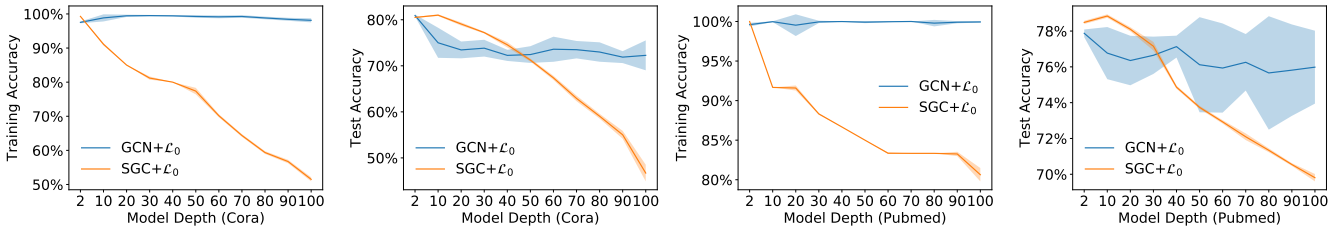

Figure 3: Comparison of Deep GCN and Deep SGC on *Cora* and *Pubmed*

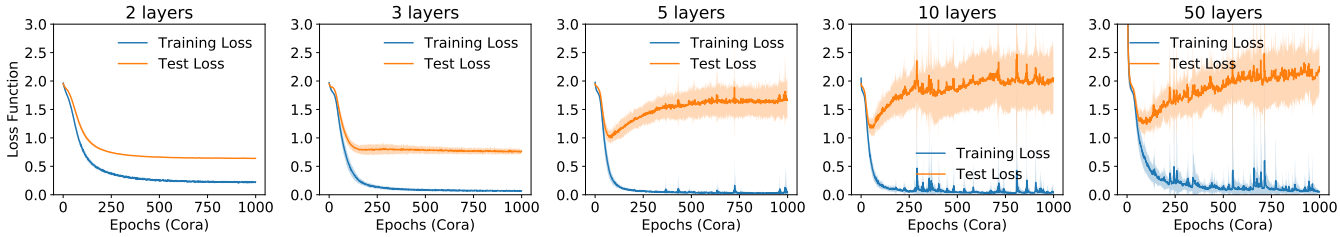

Figure 4: Training and Test Curve with 2-, 3-, 5-, 10-, 50-layer GCNs on *Cora*

activation function. All the deep models (with more than 3 hidden layers) are implemented with skip-connection (Kipf and Welling 2016; He et al. 2016), since skip-connection (also called residual connection) are necessary to prevent gradient exploding/vanishing in deep architectures, and we do not consider them as new models. We add the output of l-th layer to $(l+2)$-th layer after the *ReLU* function. Three benchmark datasets (*Cora*, *Citeseer*, *Pubmed*) are considered. We follow the same experimental settings from (Kipf and Welling 2016) and show the basic statistics of datasets in Table. 1. All the experiments are conducted with *PyTorch 1.4.0* for 20 times and mainly finished in a Linux server with 64GB memory, 32 CPUs and two GTX-2080 GPUs. Details and additional experiments could be found in Appendix H. All the datasets, demo, baseline, main codes, visualization scripts can be found in code appendix.

5.1 Overfitting in Deep GCNs

The performance of GCNs is known to decrease with increasing number of layers, for which, a common explanation is *oversmoothing* (Li, Han, and Wu 2018). In Sec. 4, we already analyze that deep GCNs can learn to anti-oversmooth. Empirically, this section further conjecture that *overfitting might be the major reason for the drop of performance in deep GCNs*. We start with a comparison between the accuracy curve of GCN and SGC, and the latter one shows a typical oversmoothing pattern.

Oversmoothing or Not. SGC (Wu et al. 2019) is a linear version of GCN without activation function. We show the accuracy curves of deep GCNs (with $\mathcal{L}_0$) and deep SGC (with $\mathcal{L}_0$) on *Cora* and *Pubmed* for various model depths in Fig. 3. An analysis of oversmoothing for SGC can be found in Appendix E. It is interesting that the accuracy of SGC decreases rapidly (Oono and Suzuki 2019) with more graph convolutions either for training or test. This is a strong indicator of oversmoothing. The performance of the GCN model is not as good as SGC soon after 2 layers (because of overfitting possibly), but it stabilizes at a high accuracy even as the model goes very deep, which presents a non-oversmoothing pattern.

Loss Function vs Depth. For a further investigation, we compute the training and test loss of the vanilla GCN models (with residual connection) on *Cora, Citeseer* and *Pubmed*. The loss curve of 2-, 3-, 5-, 10-, 50-layer GCNs with 1000 epochs on *Cora* is reported in Figure 4 (reader could find similar *Citeseer* and *Pubmed* figures in Appendix H). We notice that for shallow models (2- or 3-layer GCN), both the training and the test curve goes down with more epochs. However in deeper GCNs, the training curve almost hits the ground and the test curve first decreases and then increases gradually with more epochs (note that while the test loss increases, the test accuracy remains stable, reader could refer to Appendix H). We therefore conclude that *overfitting* might be the major factor that leads to the performance drop in deep GCNs. Note that the test loss is almost horizontal for a 3-layer GCN, so we think 3 (or 4) layers might be a separation between overfitting or not, which is consistent with the common understandings that 2 or 3 layer-GCN works better than deep GCNs in most cases.

5.2 Mean-subtraction for GCNs

In this section, we evaluate the efficacy of the *mean-subtraction* trick and compare it with vanilla GCNs (Kipf and Welling 2016), *PairNorm* (Zhao and Akoglu 2019) and the commonly used *BatchNorm* (Ioffe and Szegedy 2015). The four models have the same configurations, such as the number of layers (64), epochs (400), learning rate (0.01), and hidden units (16). They differ in how to transform the

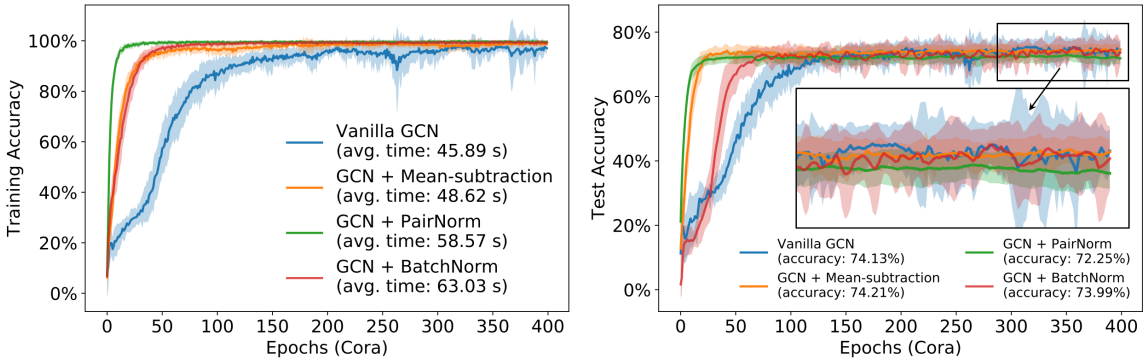

Figure 5: Comparison of Different Tricks in Training Deep GCNs on *Cora*.

Table 2: Performance vs Neighborhood Aggregation Weight (2-layer and 32-layer) on *Cora*

Accuracy (%)		$w(\eta)$=0	0.1	0.2	0.5	1.0	2	5	10	20	50	100
2-layer	training	92.66	95.67	96.32	96.05	95.33	94.54	93.44	93.30	92.82	92.86	92.98
	test	50.75	74.99	78.11	80.38	81.23	80.90	79.82	80.01	80.50	79.77	79.10
32-layer	training	95.02	99.49	99.58	99.35	98.69	98.10	98.84	98.83	98.81	98.76	98.83
	test	39.93	72.53	73.59	73.65	74.03	75.11	74.96	75.08	75.49	74.64	74.74

layerwise feature matrices. *Mean-subtraction* is to subtract the mean feature value before each convolution layer, and *PairNorm* will add a re-scaling step on the top. These two tricks do not include additional parameters. *BatchNorm* includes more parameters for each layer, which learns the mean and variance of feature representation. The experiment is conducted for all three datasets. In this section, we plot for *Cora* (in Fig. 5) due to space limitation. Readers could find similar *Citeseer* and *Pubmed* curves in Appendix H.

Analysis. We report the *Cora* result in Fig. 5, with average time consumption for 400 epochs and average accuracy of last 50 epochs listed in the figures. After all, *mean-subtraction*, *PairNorm* and *BatchNorm* all help to improve the training process: (i) fit training data well (see high training accuracy); (ii) give fast convergence. Compared to *BatchNorm*, *mean-subtraction* provides a robust/stable training curve (small variance) with less executed time. Compared to *PairNorm*, our *mean-subtraction* outputs a higher accuracy efficiently in test data. We think that additional re-scaling step in *PairNorm* might cause severe overfitting problem. In sum, *mean-subtraction* not only speeds up the model convergence but also retains the same expressive power. It is an ideal trick for training deep GCNs.

5.3 Weight in GCN Neighborhood Aggregation

In Sec. 3, we choose the learning rate $\eta = \frac{\mathrm{Tr}(X^\top X)}{2-\frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}}$. However, a different learning rate does lead to different weights $w(\eta)$ of neighbor information aggregation (we show that $w(\eta)$ is a monotonically increasing function in Appendix C). There are also some efforts on trying different ways to aggregate neighbor information (Veličković et al. 2017; Hamilton, Ying, and Leskovec 2017; Rong et al. 2019; Chen, Ma, and Xiao 2018). In this section, we consider the form "$I+w(\eta)A$" with $w(\eta) \in [0, \infty)$ and exploit a group of convolution operators by their normalized version. GCN with normalized $I + w(\eta)A$ is named as η-GCN. We evaluate this operator group on *Cora* and list the experimental results in Table. 2

Analysis. We conclude that when $w(\eta)$ is small (i.e., η is small), which means the gradient of $\mathcal{L}_{reg}$ does not contribute much to the end effect, η-GCN is more of a DNN. As $w(\eta)$ increases, a significant increase in model performance is initially observed. When $w(\eta)$ exceeds some threshold, the accuracy saturates, remaining high (or maybe decreases slightly) even as we increase $w(\eta)$ substantially. We conclude that for the widely used shallow GCNs, the common choice of weight $w(\eta) = 1$, which means a learning rate, $\eta = \frac{\mathrm{Tr}(X^\top X)}{2-\frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}}$, is large enough to include the $\mathcal{L}_{reg}$ gradient descent effect and small enough to avoid the drop in accuracy. For a deeper GCN model, larger weight (> 1.0) is preferable. To find the best weight of neighbor averaging, further inspection is needed in future work.

6 Conclusion

In this work, we reformulate the graph convolutional networks (GCNs) from MLP-based graph regularization algorithm. Based on that, we analyze the training process of deep GCNs and provide a new understanding: *deep GCNs have the power to learn anti-oversmoothing by nature, and overfitting might be the major reason for the performance drop when model goes deep*. We further propose a cheap but effective *mean-subtraction* trick to accelerate the training of deep GCNs. Extensive experiments are presented to verify our theories and provide more practical insights.

References

Ando, R. K.; and Zhang, T. 2007. Learning on graph with Laplacian regularization. In *Advances in neural information processing systems*, 25–32.

Belkin, M.; and Niyogi, P. 2002. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in neural information processing systems*, 585–591.

Belkin, M.; and Niyogi, P. 2003. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation* 15(6): 1373–1396.

Chen, D.; Lin, Y.; Li, W.; Li, P.; Zhou, J.; and Sun, X. 2019. Measuring and Relieving the Over-smoothing Problem for Graph Neural Networks from the Topological View. *arXiv preprint arXiv:1909.03211* .

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

Chen, S.; Varma, R.; Sandryhaila, A.; and Kovačević, J. 2015. Discrete Signal Processing on Graphs: Sampling Theory. *IEEE transactions on signal processing* 63(24): 6510–6523.

Chen, Y.; Wu, L.; and Zaki, M. J. 2019. Deep Iterative and Adaptive Learning for Graph Neural Networks. *arXiv preprint arXiv:1912.07832* .

Chiang, W.-L.; Liu, X.; Si, S.; Li, Y.; Bengio, S.; and Hsieh, C.-J. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 257–266.

Chung, F. R.; and Graham, F. C. 1997. *Spectral graph theory*. 92. American Mathematical Soc.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, 3844–3852.

Dehmamy, N.; Barabási, A.-L.; and Yu, R. 2019. Understanding the Representation Power of Graph Neural Networks in Learning Graph Topology. In *Advances in Neural Information Processing Systems*, 15387–15397.

Deng, Z.; Dong, Y.; and Zhu, J. 2019. Batch virtual adversarial training for graph convolutional networks. *arXiv preprint arXiv:1902.09192* .

Duvenaud, D. K.; Maclaurin, D.; Iparraguirre, J.; Bombarell, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. P. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, 2224–2232.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1263–1272. JMLR. org.

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, 1024–1034.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Ioffe, S.; and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* .

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

Klicpera, J.; Bojchevski, A.; and Günnemann, S. 2018. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997* .

Li, G.; Muller, M.; Thabet, A.; and Ghanem, B. 2019. Deepgcns: Can gcns go as deep as cnns? In *Proceedings of the IEEE International Conference on Computer Vision*, 9267–9276.

Li, Q.; Han, Z.; and Wu, X.-M. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Liu, Q.; Nickel, M.; and Kiela, D. 2019. Hyperbolic graph neural networks. In *Advances in Neural Information Processing Systems*, 8228–8239.

Loukas, A. 2019. What graph neural networks cannot learn: depth vs width. *arXiv preprint arXiv:1907.03199* .

NT, H.; and Maehara, T. 2019. Revisiting Graph Neural Networks: All We Have is Low-Pass Filters. *arXiv preprint arXiv:1905.09550* .

Oono, K.; and Suzuki, T. 2019. Graph Neural Networks Exponentially Lose Expressive Power for Node Classification. *arXiv preprint cs.LG/1905.10947* .

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

Rong, Y.; Huang, W.; Xu, T.; and Huang, J. 2019. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *International Conference on Learning Representations*.

Sandryhaila, A.; and Moura, J. M. 2013. Discrete signal processing on graphs. *IEEE transactions on signal processing* 61(7): 1644–1656.

Shaham, U.; Stanton, K.; Li, H.; Nadler, B.; Basri, R.; and Kluger, Y. 2018. Spectralnet: Spectral clustering using deep neural networks. *arXiv preprint arXiv:1801.01587* .

Smola, A. J.; and Kondor, R. 2003. Kernels and regularization on graphs. In *Learning theory and kernel machines*, 144–158. Springer.

Tenenbaum, J. B.; De Silva, V.; and Langford, J. C. 2000. A global geometric framework for nonlinear dimensionality reduction. *science* 290(5500): 2319–2323.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* .

Veličković, P.; Fedus, W.; Hamilton, W. L.; Liò, P.; Bengio, Y.; and Hjelm, R. D. 2018. Deep graph infomax. *arXiv preprint arXiv:1809.10341* .

Von Luxburg, U. 2007. A tutorial on spectral clustering. *Statistics and computing* 17(4): 395–416.

Weston, J.; Ratle, F.; Mobahi, H.; and Collobert, R. 2012. Deep learning via semi-supervised embedding. In *Neural networks: Tricks of the trade*, 639–655. Springer.

Wu, F.; Zhang, T.; Souza Jr, A. H. d.; Fifty, C.; Yu, T.; and Weinberger, K. Q. 2019. Simplifying graph convolutional networks. *arXiv preprint arXiv:1902.07153* .

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2018a. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* .

Xu, K.; Li, C.; Tian, Y.; Sonobe, T.; Kawarabayashi, K.-i.; and Jegelka, S. 2018b. Representation learning on graphs with jumping knowledge networks. *arXiv preprint arXiv:1806.03536* .

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

Ying, Z.; You, J.; Morris, C.; Ren, X.; Hamilton, W.; and Leskovec, J. 2018. Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*, 4800–4810.

Zhao, L.; and Akoglu, L. 2019. PairNorm: Tackling Oversmoothing in GNNs. *arXiv preprint arXiv:1909.12223* .

Zhou, J.; Cui, G.; Zhang, Z.; Yang, C.; Liu, Z.; Wang, L.; Li, C.; and Sun, M. 2018. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434* .

Zhu, X.; and Ghahramani, Z. 2002. Learning from labeled and unlabeled data with label propagation. Technical report, Carnegie Mellon University.

A $\mathcal{L}_{reg}$ and Spectral Clustering

Graph Regularizer $\mathcal{L}_{reg}$. $\mathcal{L}_{reg}$ is commonly formulated by Dirichlet energy, $\mathcal{L}_{reg} = \frac{1}{2}\,\mathrm{Tr}\left(H^\top \Delta H\right)$, where $f(\cdot)$ is a mapping from the input feature X to low-dimensional representation $H = f(X)$. To minimize $\mathcal{L}_{reg}$, this paper adds constraint on the magnitude of H, i.e., $\|H\|_F^2 = C \in \mathbb{R}$, which gives,

$$\min_H \ \frac{1}{2}\,\mathrm{Tr}\left(H^\top \Delta H\right),\ \textit{subject to const.}\ \|H\|_F^2. \tag{15}$$

Spectral Clustering. Given a graph with binary adjacency matrix A, a partition of node set V into k set could be written as $P_1, P_2, \ldots, P_k$ in graph theory. For normalized spectral clustering, the k indicator vectors is written as $h_i = (h_i^1, \ldots, h_i^n)$, where h_i^j represents the affiliation of node j in class set P_i and $vol(P_i) = \sum_{v_j \in P_i} d_j$ is the volume.

$$h_i^j = \begin{cases} \frac{1}{\sqrt{vol(P_i)}}, & v_j \in P_i \\ 0, & otherwise \end{cases} \tag{16}$$

The $H = [h_i^j]_{i=1..k, j=1..n}$ is a matrix containing these k indicator vectors as columns. For each row of H, there is only one non-empty entry, implying $h_i^\top h_j = 0,\ \forall i \neq j$. Let us revisit the *Normalized Cut* of a graph for a partition $P_1, P_2, \ldots, P_k$.

$$\begin{aligned}
\mathrm{Ncut}(P_1, P_2, \ldots, P_k) &= \sum_{i=1}^{k} \frac{cut(P_i, \bar{P}_i)}{vol(P_i)} \\
&= \frac{1}{2}\sum_{i=1}^{k} \frac{\sum_{v_j \in P_i, v_t \notin P_i} A_{jt} + \sum_{v_j \notin P_i, v_t \in P_i} A_{jt}}{\sqrt{vol(P_i)}\sqrt{vol(P_i)}} \\
&= \frac{1}{2}\sum_{i=1}^{k} \left(\sum_{v_j \in P_i, v_t \notin P_i} \left(\frac{1}{\sqrt{vol(P_i)}}\right)^2 + \sum_{v_j \notin P_i, v_t \in P_i} \left(\frac{1}{\sqrt{vol(P_i)}}\right)^2 \right) \\
&= \frac{1}{2}\sum_{i=1}^{k}\sum_{j,t} (h_i^j - h_i^t)^2 = \frac{1}{2}\sum_{i=1}^{k} h_i^\top L h_i = \frac{1}{2}\mathrm{Tr}(H^\top L H).
\end{aligned} \tag{17}$$

Also, H satisfies $H^\top D H = I$. When the discreteness condition is relaxed and H is substitute by $H = D^{-\frac{1}{2}}U$, the normalized graph cut problem (normalized spectral clustering) is relaxed into,

$$\min_U \ \frac{1}{2}\,\mathrm{Tr}\left(U^\top \Delta U\right),\ \ \textit{subject to}\ \ U^\top U = I. \tag{18}$$

This is a standard trace minimization problem which is solved by the matrix the eigen matrix of Δ. Compared to Eqn. (15), Eqn. (18) has a stronger constraints, which outputs the optimal solution irrelevant to the inputs (feature matrix X). However, Eqn. (15) only add constraints on the magnitude of H, which balances the trade-off and will give a solution induced by both the eigen matrix of Δ and the original feature X.

B Reyleigh Quotient $R(X)$

Reyleigh Quotient. The Reyleigh Quotient of a vector $x \in \mathbb{R}^m$ is the scalar,

$$R(x) = \frac{x^\top \Delta x}{x^\top x}, \tag{19}$$

which is invariant to the scaling of x. For example, $\forall\ c_1 \neq 0 \in \mathbb{R}$, we have $R(x) = R(c_1 \cdot x)$. When we view $R(x)$ as a function on m-dim variable x, it has stationary points x_i, where x_i is the eigenvector of Δ. Let us assume $\Delta x_i = \lambda_i x_i$, then the stationary value at point x_i will be exactly the eigenvalue λ_i,

$$R(x_i) = \frac{x_i^\top \Delta x_i}{x_i^\top x_i} = \frac{x_i^\top \lambda_i x_i}{x_i^\top x_i} = \lambda_i. \tag{20}$$

When x is not an eigenvector of Δ, the partial derivatives of $R(x)$ with respect to the vector coordinate x_j is calculated as,

$$\begin{aligned}
\nabla_{x_j} R(x) = \frac{\partial R(x)}{\partial x_j} &= \frac{\frac{\partial}{\partial x_j}(x^\top \Delta x)}{x^\top x} - \frac{(x^\top \Delta x)\frac{\partial}{\partial x_j}(x^\top x)}{(x^\top x)^2} \\
&= \frac{2(\Delta x)_j}{x^\top x} - \frac{(x^\top \Delta x) 2x_j}{(x^\top x)^2} = \frac{2}{x^\top x}(\Delta x - R(x)x)_j
\end{aligned} \tag{21}$$

Thus, the derivative of $R(x)$ with respect to x is collected as,

$$\nabla R(x) = \frac{2}{x^\top x}(\Delta x - R(x)x). \tag{22}$$

Minimizing R(x). Suppose $\Delta = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the normalized Laplacian matrix. Let us first consider to minimize $R(x)$ without any constraints. Since Δ is a symmetric real-valued matrix, it could be factorized by Singular Value Decomposition,

$$\Delta = U\Lambda U^\top = \sum_{i=1}^{s} u_i \lambda_i u_i^\top \tag{23}$$

where s is the rank of Δ and $0 = \lambda_1 \leq \cdots \leq \lambda_s < 2$ are the eigen values. For any non-zero vector x, it is decomposed w.r.t. the eigen space of Δ,

$$x = \epsilon + \sum_{i=1}^{s} c_i \cdot u_i \tag{24}$$

where $\{c_i\}$ is the coordinates and ϵ is a component tangent to the eigen space spanned by $\{u_i\}$. Let us consider the component of x within the eigen space and discuss ϵ later. Therefore, the Reyleigh Quotient $R(x)$ can be calculated by,

$$R(x) = \frac{x^\top \Delta x}{x^\top x} = \frac{(\sum_{i=1}^{s} c_i \cdot u_i^\top)(\sum_{i=1}^{s} u_i \lambda_i u_i^\top)(\sum_{i=1}^{s} c_i \cdot u_i)}{(\sum_{i=1}^{s} c_i \cdot u_i^\top)(\sum_{i=1}^{s} c_i \cdot u_i)} = \frac{\sum_{i=1}^{s} c_i^2 \lambda_i}{\sum_{i=1}^{s} c_i^2} \tag{25}$$

Recall the partial derivative of $R(x)$ w.r.t. x in Eqn. (22). Think about to minimize $R(x)$ by gradient descent and always consider the learning rate (the same as what we used in the main text. The factor $\frac{1}{2}$ is from that the $R(x)$ in appendix does not have the scalar $\frac{1}{2}$),

$$\eta = \frac{1}{2}\mathrm{Tr}(X^\top X)/(2 - \frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}) = \frac{1}{2} \cdot \frac{x^\top x}{2 - \frac{x^\top \Delta x}{x^\top x}} = \frac{1}{2} \cdot \frac{x^\top x}{2 - R(x)}. \tag{26}$$

The initial x is regarded as an starting point, and the next point x' is given by gradient descent,

$$x' = x - \eta \nabla R(x) = x - \frac{1}{2} \cdot \frac{x^\top x}{2 - R(x)} \frac{2}{x^\top x}(\Delta x - R(x)x) = \frac{2I - \Delta}{2 - R(x)} x. \tag{27}$$

The new Reyleigh Quotient value is,

$$R(x') = \frac{x'^\top \Delta x'}{x'^\top x'} = \frac{(\frac{2I-\Delta}{2-R(x)}x)^\top \Delta (\frac{2I-\Delta}{2-R(x)}x)}{(\frac{2I-\Delta}{2-R(x)}x)^\top (\frac{2I-\Delta}{2-R(x)}x)} = \frac{x^\top (2I-\Delta)\Delta(2I-\Delta)x}{x^\top (2I-\Delta)(2I-\Delta)x}. \tag{28}$$

The eigen properties of $2I - \Delta$ could be derived from Δ, where they have the same eigenvector, and any eigenvalue λ of Δ will adjust to be an eigenvalue $2 - \lambda$ of $2I - \Delta$. Therefore, we do further derivation,

$$R(x') = \frac{\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2 \lambda_i}{\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2}. \tag{29}$$

So far, to get the ideal effect, a final check is needed: whether the Reyleigh Quotient does decrease after the gradient descent.

$$\begin{aligned}
R(x') - R(x) &= \frac{\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2 \lambda_i}{\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2} - \frac{\sum_{i=1}^{s} c_i^2 \lambda_i}{\sum_{i=1}^{s} c_i^2} \\
&= \frac{(\sum_{i=1}^{s} c_i^2)(\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2 \lambda_i) - (\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2)(\sum_{i=1}^{s} c_i^2 \lambda_i)}{(\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2)(\sum_{i=1}^{s} c_i^2)} \\
&= \frac{\sum_{i,j} c_i^2 c_j^2 (\lambda_i - \lambda_j)(\lambda_j - \lambda_i)(4 - \lambda_i - \lambda_j)}{(\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2)(\sum_{i=1}^{s} c_i^2)} \\
&= -\frac{\sum_{i,j} c_i^2 c_j^2 (\lambda_i - \lambda_j)^2 (4 - \lambda_i - \lambda_j)}{(\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^2)(\sum_{i=1}^{s} c_i^2)} < 0
\end{aligned} \tag{30}$$

Also, we show the asymptotic property of $R(x)$ in gradient descent,

$$\lim_{t\to\infty} R(x^{(t)}) = \lim_{t\to\infty} \frac{\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^{2t} \lambda_i}{\sum_{i=1}^{s} c_i^2 (2-\lambda_i)^{2t}} = \frac{c_2^2 \lambda_2}{c_1^2} \cdot \lim_{t\to\infty} \left(\frac{2-\lambda_2}{2-\lambda_1}\right)^{2t} = 0^+ \tag{31}$$

where $x^{(t)}$ is the t-th new point given by gradient descent. So far, we finish the proof of well-definedness of gradient descent with the $\eta = \frac{x^\top x}{2 - \frac{x^\top \Delta x}{x^\top x}} = \frac{\mathrm{Tr}(X^\top X)}{2 - \frac{\mathrm{Tr}(X^\top \Delta X)}{\mathrm{Tr}(X^\top X)}}$.

Remark 1. In fact, as stated above, $R(x)$ is invariant to the scaling of x, so we could scale x on its magnitude, i.e., making $\|x\| = c \in \mathbb{R}^+$ as a constraint during the gradient descent iteration, all the properties and results still hold.

Remark 2. In the main text, instead of using a vector x, we use a feature matrix X and define our Reyleigh Quotient by $R(X) = \frac{\text{Tr}(X^\top \Delta X)}{\text{Tr}(X^\top X)}$. In fact, different feature channels of X could be viewed as independent vector signal $x_i \in \mathbb{R}^m$ and for each channel, the same gradient descent analysis is applied. Therefore, we finish the detailed proof for our formulation in the main text, which is of the following form,

$$\min\ R(X),\ \textit{subject to const.}\ \|X\|_F^2. \tag{32}$$

C Learning Rate η and Neighbor Averaging Weight $w(\eta)$

We show the relation of learning rate η and neighbor averaging weight $w(\eta)$ in this section (to make the derivation consistent with the main text, $\nabla R(x)$ does not have factor 2).

$$X' = X - \eta \nabla R(X) = X - \eta \frac{1}{\text{Tr}(X^\top X)}(\Delta X - R(X)X) \tag{33}$$

$$= (\frac{R(X) - 1}{\text{Tr}(X^\top X)} \cdot \eta + 1)(I + \frac{\eta}{(R(x) - 1) \cdot \eta + \text{Tr}(X^\top X)} \cdot D^{-\frac{1}{2}} A D^{-\frac{1}{2}})X \tag{34}$$

The first multiplier $(\frac{R(X)-1}{\text{Tr}(X^\top X)} \cdot \eta + 1)$ will be absorbed into the parameter matrices. Thus, we have,

$$w(\eta) = \frac{\eta}{(R(x) - 1) \cdot \eta + \text{Tr}(X^\top X)} = \frac{1}{R(x) - 1 + \frac{\text{Tr}(X^\top X)}{\eta}}, \tag{35}$$

According to the formulation, $w(\eta)$ is a monotonically increasing function on variable η and is valid when $w(\eta) > 0$. Therefore, when $R(x) \geq 1$, the domain of the function is $\eta \in [0, \infty)$ and when $R(x) < 1$ (we know from Eqn. (31) that $R(x) \to 0^+$), the domain of the function is bounded, $\eta \in [0, \frac{\text{Tr}(X^\top X)}{1 - R(x)})$.

Remark 3. Note that the choice in this paper, $\eta = \frac{\text{Tr}(X^\top X)}{2 - \frac{\text{Tr}(X^\top \Delta X)}{\text{Tr}(X^\top X)}}$, always lies in the valid domain for $\forall x \in \mathbb{R}^m$. Also, in the valid domain, with respect to the change of η, $w(\eta)$ can vary in the range $[0, \infty)$ monotonically.

D Proof of Theorem 1

Proof. Given any non-zero signal $x \in \mathbb{R}^n$ and a symmetric matrix $A \in \mathbb{R}^{n \times n}$ (with non-negative eigenvalues), we factorize them in the eigenspace,

$$A = U \Lambda U^\top = \sum_{i=1}^{s} u_i \lambda_i u_i^\top \quad \text{and} \quad x = \epsilon + \sum_{i=1}^{s} c_i \cdot u_i \tag{36}$$

where S is of rank $s \in \mathbb{N}^+$, $U = [u_i]_{i=1}^s$ and $\Lambda = diag(\lambda_1, \cdots, \lambda_s)$ are eigen matrices. $\{c_i \in \mathbb{R}\}_{i=1}^s$ are coordinates of x in the eigenspace and $\epsilon \in \mathbb{R}$ is a component tangent to the eigenspace. In a k-layer SGC, the effect of graph convolution is the same as applying Laplacian smoothing k times,

$$A^k x = \left(\sum_{i=1}^{s} u_i \lambda_i u_i^\top \right)^k \left(\epsilon + \sum_{i=1}^{s} c_i \cdot u_i \right) = \sum_{i=1}^{s} c_i \lambda_i^k u_i. \tag{37}$$

Suppose λ_1 is the largest eigenvalue and $c_1 \neq 0$. We go with infinite number of layers and then have $\lim_{k \to \infty} S^k x \propto u_1$, which means the output is unrelated to the input features x. □

E Analysis of SGC

SGC was proposed in (Wu et al. 2019), with the hypothesis that the non-linear activation is not critical while the majority of the benefit arises from the local averaging A. The authors directly remove the activation function and proposed a linear "L-layer" model, where $\prod W^{(l)}$ has collapsed into a single W.

$$f(X) = A\left(A(\cdots)W^{(L-1)}\right)W^{(L)} = A^L X W. \tag{38}$$

This model explicitly disentangles the dependence of STEP1 and STEP2. We similarly formulate the SGC model in the form of two-step optimization,

$$\text{STEP1: } \min_X \mathcal{L}_{reg}(X) \quad \text{and} \quad \text{STEP2: } \min_W \mathcal{L}_0(W) \tag{39}$$

From the two-step optimization form, SGC is essentially conducting gradient descent algorithm L times in STEP1. In STEP2, SGC model will seek to minimize L_0 on the basis of the oversmoothed features. The independence between STEP1 and STEP2 accounts for the oversmoothing issue, which cannot be mitigated during training.

F Mean-subtraction

Background. In the main text, we start with one of the most popular convolution operator A_{rw} and its largest eigenvector $u_1 = \mathbf{1} \in \mathbb{R}^n$. Let us use a simplified notation $\bar{u_1} = \frac{u_1}{\|u_1\|}$. Given any non-zero $x \in \mathbb{R}^n$, the proposed mean-subtraction has the following form,

$$x_{new} \leftarrow x - x_{mean} = x - \frac{\mathbf{1}\mathbf{1}^\top x}{n} = x - \langle x, \bar{u_1} \rangle \cdot \bar{u_1} \tag{40}$$

There are some facts from spectral graph theory that

- A_{sys} and A_{rw} have the same eigenvalues, $1 = \lambda_1 \geq \cdots \lambda_s > 0$.
- If u is an eigenvector of A_{sys}, i.e., $A_{sys}u = \lambda u$. Then $\tilde{D}^{-\frac{1}{2}}u$ is an eigenvector of A_{rm} with the same eigenvalue, i.e., $A_{rm}\tilde{D}^{-\frac{1}{2}}u = \lambda \tilde{D}^{-\frac{1}{2}}u$.
- The eigenvector associated with the largest eigenvalue $\lambda_1 = 1$ of A_{sys} is $u_1 = \tilde{D}^{\frac{1}{2}}$, while for A_{rw}, it is $A_{rw}\mathbf{1} = \lambda_1 \mathbf{1} = \mathbf{1}$.

Mean-subtraction for A_{sys}. Let first discuss the graph convolution operator A_{sys},

$$A_{sys} = \tilde{D}^{-\frac{1}{2}}(I + A)\tilde{D}^{-\frac{1}{2}} = U\Lambda U^\top = \sum_{i=1}^{s} u_i \lambda_i u_i^\top \tag{41}$$

and the signal x as

$$x = \epsilon + \sum_{i=1}^{s} c_i \cdot u_i \tag{42}$$

Then we apply the Laplacian smoothing k times,

$$\begin{aligned} A_{sys}^k x &= \left(\sum_{i=1}^{s} u_i \lambda_i u_i^\top\right)^k \left(\epsilon + \sum_{i=1}^{s} c_i \cdot u_i\right) \\ &= \left(\sum_{i=1}^{s} u_i \lambda_i^k u_i^\top\right) \left(\epsilon + \sum_{i=1}^{s} c_i \cdot u_i\right) = \sum_{i=1}^{s} c_i \lambda_i^k u_i. \end{aligned} \tag{43}$$

which tells that $\lim_{k\to\infty} A_{sym}^k x \propto u_1 = \tilde{D}^{\frac{1}{2}}\mathbf{1}$. The mean-subtraction trick on A_{sym} is of a factor $\tilde{D}^{\frac{1}{2}}$ (suppose mapping $f(x) = \tilde{D}^{\frac{1}{2}}x$ and inverse mapping $f^{-1}(x) = \tilde{D}^{-\frac{1}{2}}x$),

$$x_{new} \leftarrow f^{-1}(f(x) - f(x_{mean})) = \tilde{D}^{-\frac{1}{2}}(1 - \frac{\mathbf{1}\mathbf{1}^\top}{n})\tilde{D}^{\frac{1}{2}}x = x - \frac{\tilde{D}^{\frac{1}{2}}\mathbf{1}\mathbf{1}^\top\tilde{D}^{-\frac{1}{2}}x}{n}. \tag{44}$$

Therefore, after one layer of mean-subtraction, the signal x would be,

$$\begin{aligned}
x_{new} &\leftarrow f^{-1}(f(x) - f(x_{mean})) \\
&= \tilde{D}^{-\frac{1}{2}}(1 - \frac{\mathbf{1}\mathbf{1}^\top}{n})\tilde{D}^{\frac{1}{2}}\left(\epsilon + \sum_{i=1}^{s} c_i \cdot u_i\right) \\
&= \tilde{D}^{-\frac{1}{2}}(1 - \frac{\mathbf{1}\mathbf{1}^\top}{n})\tilde{D}^{\frac{1}{2}}\sum_{i=1}^{s} c_i \cdot u_i \\
&= \sum_{i=1}^{s} \tilde{D}^{-\frac{1}{2}}(1 - \frac{\mathbf{1}\mathbf{1}^\top}{n})\tilde{D}^{\frac{1}{2}} c_i \cdot u_i \\
&= \sum_{i=1}^{s} c_i \cdot u_i - \sum_{i=1}^{s} \tilde{D}^{-\frac{1}{2}}\frac{\mathbf{1}\mathbf{1}^\top}{n}\tilde{D}^{\frac{1}{2}} c_i \cdot u_i \\
&= \sum_{i=1}^{s} c_i \cdot u_i - \sum_{i=1}^{1} c_i \cdot u_i = \sum_{i=2}^{s} c_i \cdot u_i
\end{aligned} \tag{45}$$

which eliminate the dominant effect of u_1.

Mean-subtraction for A_{rw}. Then for the graph convolution operator A_{rw}, we could do the similar decomposition,

$$\begin{aligned}
A_{rw} &= \tilde{D}^{-1}(I + A) \\
&= \tilde{D}^{-\frac{1}{2}}\left(\tilde{D}^{-\frac{1}{2}}(I + A)\tilde{D}^{-\frac{1}{2}}\right)\tilde{D}^{\frac{1}{2}} \\
&= \tilde{D}^{-\frac{1}{2}}\left(U\Lambda U^\top\right)\tilde{D}^{\frac{1}{2}} \\
&= \tilde{D}^{-\frac{1}{2}}\left(\sum_{i=1}^{s} u_i\lambda_i u_i^\top\right)\tilde{D}^{\frac{1}{2}}
\end{aligned} \tag{46}$$

and for the signal x into $\{\tilde{D}^{-\frac{1}{2}}u\}$ space as

$$x = \epsilon + \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}}u_i \tag{47}$$

Similar we apply the Laplacian smoothing k times,

$$\begin{aligned}
A_{rw}^k x &= \left(\tilde{D}^{-\frac{1}{2}}\left(\sum_{i=1}^{s} u_i\lambda_i u_i^\top\right)\tilde{D}^{\frac{1}{2}}\right)^k \left(\epsilon + \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}}u_i\right) \\
&= \underbrace{\left(\tilde{D}^{-\frac{1}{2}}\left(\sum_{i=1}^{s} u_i\lambda_i u_i^\top\right)\tilde{D}^{\frac{1}{2}}\right)\cdots\left(\tilde{D}^{-\frac{1}{2}}\left(\sum_{i=1}^{s} u_i\lambda_i u_i^\top\right)\tilde{D}^{\frac{1}{2}}\right)}_{k\ terms}\left(\epsilon + \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}}u_i\right) \\
&= \tilde{D}^{-\frac{1}{2}}\left(\sum_{i=1}^{s} u_i\lambda_i^k u_i^\top\right)\tilde{D}^{\frac{1}{2}}\left(\epsilon + \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}}u_i\right) = \sum_{i=1}^{s} c_i\lambda_i^k\tilde{D}^{-\frac{1}{2}}u_i.
\end{aligned} \tag{48}$$

which tells that $\lim_{k\to\infty} A_{rw}^k x \propto \tilde{D}^{-\frac{1}{2}}u_1 = \mathbf{1}$. The mean-subtraction trick on A_{rw} is

$$x_{new} \leftarrow x - x_{mean} = x - \frac{\mathbf{1}\mathbf{1}^\top x}{n} = x - \langle x, \bar{u_1}\rangle \cdot \bar{u_1} \tag{49}$$

Therefore, after one layer of mean-subtraction, the signal x would be,

$$\begin{aligned}
x_{new} &\leftarrow x - x_{mean} \\
&= (1 - \frac{\mathbf{1}\mathbf{1}^\top}{n}) \left(\epsilon + \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i \right) \\
&= (1 - \frac{\mathbf{1}\mathbf{1}^\top}{n}) \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i \\
&= \sum_{i=1}^{s} (1 - \frac{\mathbf{1}\mathbf{1}^\top}{n}) c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i \\
&= \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i - \sum_{i=1}^{s} \frac{\mathbf{1}\mathbf{1}^\top}{n} c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i \\
&= \sum_{i=1}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i - \sum_{i=1}^{1} c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i = \sum_{i=2}^{s} c_i \cdot \tilde{D}^{-\frac{1}{2}} u_i
\end{aligned} \tag{50}$$

which eliminate the dominant effect of $\tilde{D}^{-\frac{1}{2}} u_1 = \mathbf{1}$.

Remark 4. So far, we discuss the one layer mean-subtraction for both A_{sys} and A_{rw} and also the powering effect of A_{sys} and A_{rw} on arbitrary signal x (c_1 is non-zero). Although we have show that one layer of mean-subtraction could eliminate the dominant eigenvector (once and for all). However, in the main text, we discuss that in the non-linear deep GCN architecture, which means after the ReLU activation function, the effect of dominant eigenvector may still appear. Therefore, we need mean-subtraction layer after applying activation function and iteratively eliminate u_1 or $\tilde{D}^{-\frac{1}{2}} u_1$. Due to the powering effect, they will finally approximate the Fiedler vector,

$$\lim_{k\to\infty} [A_{rw}^k x]_{mean-subtraction} \propto \tilde{D}^{-\frac{1}{2}} u_2 \quad \text{and} \quad \lim_{k\to\infty} [A_{sys}^k x]_{mean-subtraction} \propto u_2 \tag{51}$$

G *Karate* Demonstration

The Cosine Similarity. Suppose the feature matrix after the l-th layer is $X^{(l)} \in \mathbb{R}^{n\times m}$,

$$X^{(l)} = \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \\ x_{n1} & & x_{nm} \end{bmatrix} \tag{52}$$

we use $X_i,\ i = 1..n$ to denote the rows of $X^{(l)}$ and use $X^j,\ j = 1..m$ to denote the cols of $X^{(l)}$. $\{X_i\}$ is row vector and $\{X^j\}$ is column vector. The feature-wise smoothing (cosine similarity) is given by the averaging absolute value of the following matrix,

$$\text{avg} \begin{bmatrix} \frac{X^{1\top}X^1}{\|X^1\|\|X^1\|} & \frac{X^{1\top}X^2}{\|X^1\|\|X^2\|} & \cdots & \frac{X^{1\top}X^m}{\|X^1\|\|X^m\|} \\ \frac{X^{2\top}X^1}{\|X^2\|\|X^1\|} & \frac{X^{2\top}X^2}{\|X^2\|\|X^2\|} & \cdots & \frac{X^{2\top}X^m}{\|X^2\|\|X^m\|} \\ \vdots & \vdots & \ddots & \\ \frac{X^{m\top}X^1}{\|X^m\|\|X^1\|} & \frac{X^{m\top}X^2}{\|X^m\|\|X^2\|} & & \frac{X^{m\top}X^m}{\|X^m\|\|X^m\|} \end{bmatrix} \tag{53}$$

The maximum possible score is 1 if all the entry are either 1 or -1, which means all of they are entirely on the same direction.

Similarly, the node-wise smoothing (cosine similarity) is given by the averaging absolute value of the following matrix,

$$\text{avg} \begin{bmatrix} \frac{X_1X_1^\top}{\|X_1\|\|X_1\|} & \frac{X_1X_2^\top}{\|X_1\|\|X_2\|} & \cdots & \frac{X_1X_n^\top}{\|X_1\|\|X_n\|} \\ \frac{X_2X_1^\top}{\|X_2\|\|X_1\|} & \frac{X_2X_2^\top}{\|X_2\|\|X_2\|} & \cdots & \frac{X_2X_n^\top}{\|X_2\|\|X_n\|} \\ \vdots & \vdots & \ddots & \\ \frac{X_nX_1^\top}{\|X_n\|\|X_1\|} & \frac{X_nX_2^\top}{\|X_n\|\|X_2\|} & & \frac{X_nX_n^\top}{\|X_n\|\|X_n\|} \end{bmatrix} \tag{54}$$

The maximum possible score is 1 if all the entry are either 1 or -1, which means all of they are entirely on the same direction.

H Experimental Details and More

H.1 Overfitting in Deep GCNs

The configuration of the model is as follows: random seed (42), epochs (400), learning rate (0.01), optimizer (Adam with $5e-4$ weight decay in parameters), hidden units (16), dropout rate (0.5). For the **first experiment**, we run the GCN model for depth 2, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 and run SGC for A^k, where k =2, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. Both models are under the supervision of $\mathcal{L}_0$. This experiment is conducted for 20 times on *Cora* and *Pubmed*, and we draw the mean value and standard deviations in the figure. As stated in the experiment section, we apply basic residual connection between l-th and $(l+2)$-th layer after the activation function. We find that the results are similar when adding the residual connection before the activation functions. For the **second experiment**, we run GCN models for depth from 2$\sim$50 on *Cora, Citeseer, Pubmed* for 20 times with 1000 epochs. The overall figure is shown in Figure 8. We plot the train and test loss curves (*Cora*) for 2-, 3-, 5-, 10-, 50-layer GCNs with mean value and standard deviations in the main text.

The scripts of results are included in the "./oversmooth/performance-depth/.." folder, and the script of figures is the "overfitting-vis.ipynb" after getting all the results (maybe readers need to do a minor directory change when in a different directory).

H.2 Mean-subtraction for GCNs

The configuration of the model is as follows: random seed (42), epochs (400), learning rate (0.01), optimizer (Adam with $5e-4$ weight decay in parameters), number of layers (64), hidden units (16), dropout rate (0.5). The *mean-subtraction* trick is implemented by "torch.mean" function; the *PairNorm* trick is implemented by "torch.mean" and "torch.std" function; the *BatchNorm* trick is implemented by "torch.nn.BatchNorm1d" function. We run the GCN model for 20 times on *Cora, Citeseer, Pubmed*, and we draw the mean value and standard deviations in the figure. We also apply basic residual connection here.

The script is included in the "./oversmooth/mean-subtraction/.." folder, and the script of figures is the "mean-subtraction-vis.ipynb" after getting all the results

H.3 Neighborhood Aggregation Weight in GCNs

The configuration of the model is as follows: random seed (42), epochs (400), learning rate (0.01), optimizer (Adam with $5e-4$ weight decay in parameters), number of layers (2 or 32), hidden units (16), dropout rate (0.5). We run the GCN model for weight=0, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 50.0, 100.0. This experiment is conducted on *Cora*. Also, we apply basic residual connection between l-th and $(l+2)$-th layer for 32 layers deep GCNS.

The script is included in the "./oversmooth/neighbor-aggregation-weight/.." folder.

H.4 Additional Experiments

Note that all the experiments are conducted for 20 times.

- We compare deep GCNs (with $\mathcal{L}_0$), deep SGC (with $\mathcal{L}_0$) and DNN (with $\mathcal{L}_0 + \gamma\mathcal{L}_{reg}$) on *Cora* with different depths in Figure 6;
- Additional mean-subtraction evaluations for *Citeseer, Pubmed* in Figure 7;
- Training Deep GCNs for $2 \sim 50$ layers on *Cora, Citeseer, Pubmed*. We show the training/test loss and accuracy curves in Figure 8;
- We compute the training and test loss of the vanilla GCN models of 2-, 3-, 5-, 10-, 50-layer with 1000 epochs on *Citeseer* is reported in Figure 9;
- We compute the training and test loss of the vanilla GCN models of 2-, 3-, 5-, 10-, 50-layer with 1000 epochs on *Pubmed* is reported in Figure 10.

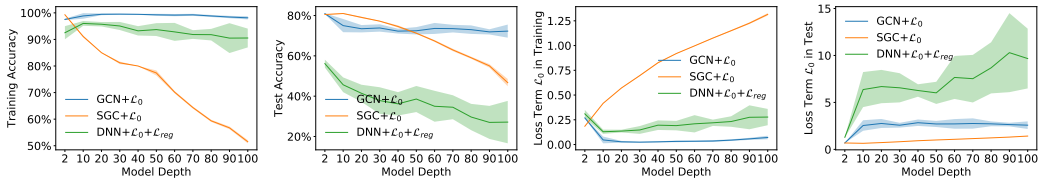

Figure 6: Comparison of Deep GCN, Deep SGC and DNN on *Cora*

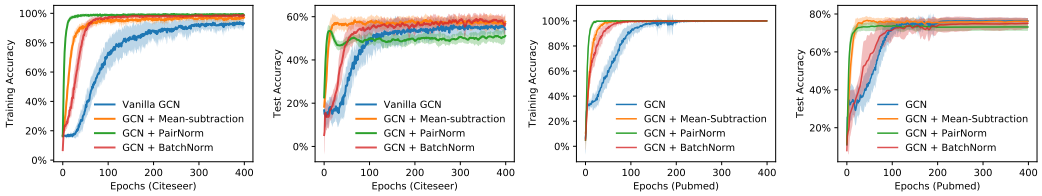

Figure 7: Comparison of Different Tricks in Training Deep GCNs on *Citeseer, Pubmed*.

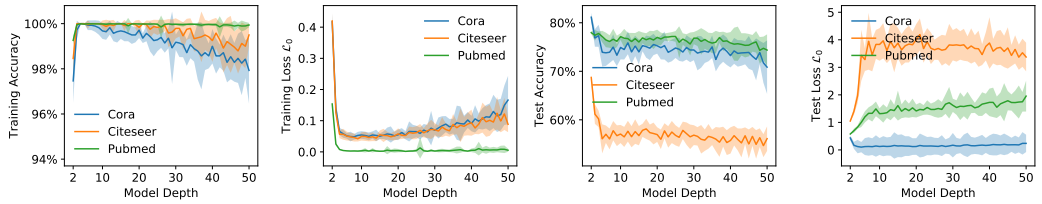

Figure 8: Training Deep GCNs for $2 \sim 50$ layers on *Cora, Citeseer, Pubmed*.

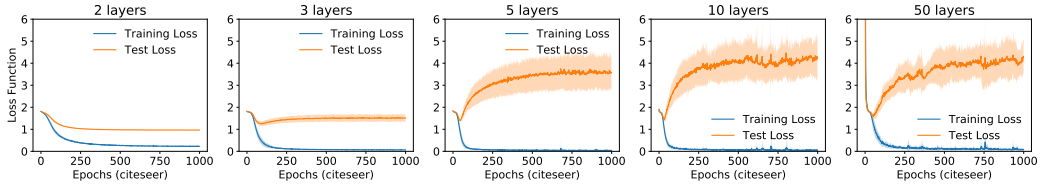

Figure 9: Training and Test Curve with 2-, 3-, 5-, 10-, 50-layer GCNs on *Citeseer*.

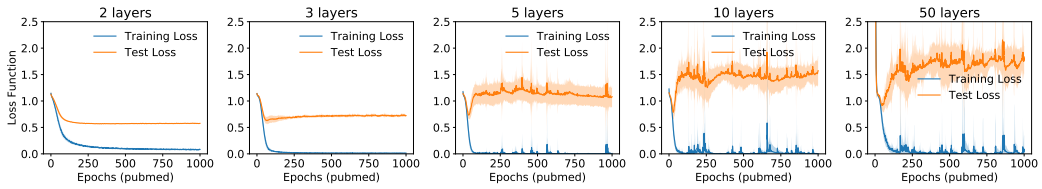

Figure 10: Training and Test Curve with 2-, 3-, 5-, 10-, 50-layer GCNs on *Pubmed*.