Understanding Graph Neural Networks via Trajectory Analysis

Ziqiao Meng[1], **Jin Dong**[2], **Zengfeng Huang**[3], **Irwin King**[1]
[1] The Chinese University of Hong Kong, [2] Mila and Mcgill University
[3] Fudan University
`ziqiao.meng@mail.utoronto.ca`

Abstract

Graph neural network (GNN) has attracted enormous attentions in machine learning communities and analyzing the architecture of GNNs have been very popular these days. Inspired by Arora's recent work, we propose utilizing "trajectory analysis" technique to see how the representation of each node evolves during training process by studying the differences of each node representation before and after a gradient update. Our studies in a simplified setting show that the update rule is quite simple and the rule indicates that the learning mechanism of GNNs could be divided into two stages: constructing similarity matrix and updating labels based on similarity matrix. Under semi-supervised learning settings, the node embedding udpate rule could even be regarded as weighted k-nearest neighbors algorithm. Last but not the least, convergence speed of different nodes will affect the model performance.

1 Introduction

Deep learning models have achieved great success on tackling image data and sequence data, while how to deal with non-euclidean data, like graph-structured data, still remains a challenging problem. Kipf and Welling [8] proposes graph convolutional networks (GCN), an architecture utilizing message passing algorithms between nodes, effectively capturing the low-dimensional representations of graphs and achieving state-of-the-art performance on the node classification task under semi-supervised learning settings. Since then, various GNNs [3, 13, 15, 18, 16, 10, 14, 17] have been presented by adjusting GCNs to adapt to different scenarios. However, with deeper and deeper explorations, two main limitations of GNNs have been revealed.

Limited Expressiveness. Morris and Xu [7, 15] states that the ability of distinguishing graph structures of current GNNs are upper bounded by Weisfeiler-Lehman (WL) graph isomorphism test [11]. In fact, GNNs are just extensions of 1-WL test by taking node features into considerations. Based on previous studies, 1-WL test and 2-WL test fail to distinguish some d-regular graphs and 3-WL test has much stronger expressiveness. Therefore, Morris, Maron and Du [7, 6, 1] enhance GNNs' performances on graph classification tasks by incorporating various graph kernel techniques. The main contribution of this stream of analysis is relating classic graph kernels [12], a well-explored tool in past years, to graph neural networks, evidently improving GNNs' performances on graph-level tasks (e.g. graph classification, graph regression). Unfortunately, introducing graph kernels to theoretical analysis of GNNs is not enough especially when we want to look into node-level or subgraph-level issues.

Shallow Structure. Another shortcoming of GNNs is shallow architecture. According to broad empirical study, GNNs could only achieve decent performances with just 2 or 3 layers most of the time and stacking more layers would result in over-smoothing issues, in other words, all nodes will converge to the same value. Clearly, this bottleneck contradicts to the compositionality principle of deep learning [4, 2] as we usually expect deeper is better. Recently, Luan [5] has proved that stacking more layers would result in low-rank learned feature matrix. If the graph dataset is connected, then the learned feature matrix tend to be rank-1 matrix. Yet, since we have already known the low-rank result before, the proof is more about verification instead of providing new insights.

In this work, we look into gradient trajectories traversed in the optimizations of GNNs. The three main contributions of this work are:

- To the best of our knowledge, this is first time analyzing weight updates of GNNs in details;
- We establish potential connections between GNNs and metric learning, which probably inspires future research on pre-training graph neural networks;
- We figure out a potential drawback that some of the nodes might converge too early during training process, which will affect the model performance.

2 Preliminaries

At the very beginning, let us look at the architecture of two-layer GCN model:

$$Y = softmax(L\ Relu(LXW_1)\ W_2),$$

where $Y \in R^{N\times C}$ denotes the final result of classifiers, $L = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \in R^{N\times N}$ denotes Laplacian Matrix, $\tilde{A} = A + I$, $A \in R^{N\times N}$ is an adjacency matrix and $I \in R^{N\times N}$ is an identity matrix. $\tilde{D} \in R^{N\times N}$ is the degree matrix of $\tilde{A}$, $X \in R^{N\times F}$ denotes input features, $W_1 \in R^{F\times D_1}$ and $W_2 \in R^{D_1\times C}$ are weight matrices, $Relu(x_i) = max(0, x)$, $softmax(x_i) = \frac{1}{Z}exp(x_i)$ with $Z = \sum_i exp(x_i)$.

The above is the original settings of GCN model. To serve our theoretical analysis, we simplify the model settings mainly in two aspects:

- we choose to put aside any non-linear activation functions, which is equivalent to simple graph convolution (SGC)[14] with multiple weight matrices.
- We substitute A for L. A here does not represent adjacency matrix. Instead, A is generalized to denote any similarity matrix, including adjacency matrix and Laplacian matrix.

We have three important reasons for doing these simplifications:

- Currently non-linear neural networks are still tough to be analyzed especially when it comes to gradient computation. Non-linearity would make the final result hard to be rearranged.
- Based on some previous findings [14, 9], removing non-linearity trivially affect model performance except for few non-linearly separable cases.
- Generalize the meaning of A is not only for notation simplicity, but for covering more cases. Many variants of GNNs just modify the node similarity matrix.

The new model setting for our theoretical analysis is as follows:

$$Y = softmax(A(AXW_1)W_2),$$

since we are going to calculate the gradient of the loss function with respect to weight matrices, let's define the semi-supervised loss function here:

$$L = -\sum_{j\in L}(\log(t_j))y_j,$$

where y_j is one-hot vector with dimension c (the number of classes), representing the ground-truth label of node j, t_j is the row vector of all class probabilities of node j. In short, this loss function is simply calculating cross entropy on labeled nodes.

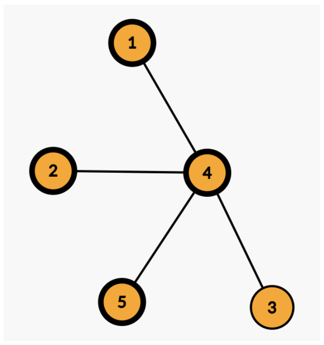

Figure 1: Assume node 1, 2 and 5 have the same label A, node 4 also has label A but currently we don't know and we want to predict it. Node 3 has label B. Now, If we give node 1, 2 and 5 perfect initialization, then they are not able to propagate their labels to node 4 according to the theorem. Consequently, node 4 would be mistakenly labeled as B due to the influence of node 3.

3 Trajectory Analysis

In this section, we will show some theoretical results by calculating the gradient of loss function with respect to W_1 and W_2.

Lemma 3.1. *For two-layer SGC, given node k, gradient update of W_2 would influence node k feature learning in the following way:*

$[AHW_2^{i+1}]_k = a_k^T HW_2^i + \sum_{j \in L} a_k^T HH^T a_j \cdot \epsilon,$

where $\epsilon = y_j^T$ - $softmax(a_j^T HW_2^i)$, $H = AXW_1^i$

Lemma 3.2. *For two-layer SGC, given node k, gradient update of W_1 would influence node k feature learning in this way:*

$[A^2XW_1^{i+1}W_2^i]_k = a_k^T AXW_1^iW_2^i + \sum_{j \in L} a_k^T AX(AX)^T a_j \cdot \epsilon \cdot (W_2^i)^T,$

where $\epsilon = y_j^T -$ *softmax* $(a_j^T AXW_1^iW_2^i)$

4 Interpretations of the Theoretical Results:

Overview of Learning Mechanism. Notice that $a_k^T HH^T a_j$ is calculating the similarity between node k and node j based on neighbor aggregations. We could regard $HH^T \in R^{n \times n}$ as a node similarity matrix constructed from the first layer. If we replace H with adjacency matrix X, then this term becomes $a_k^T XX^T a_j$, which is simply the dot product of neighbor aggregations between node k and node j. Therefore, if more neighbors node k and node j share, larger influence node j has on node k. Now we should pay attention to ϵ term. Clearly, ϵ denotes the difference between the node j's ground-truth label and node j's label predictions. If node j has ground-truth label z, we could see that only zth position of ϵ is positive while other positions are negative. In other words, node j is trying to propagate its label to node k and the effectiveness of this label propagation would be determined by its similarity with node k. Hence, we could also regard the update rule as weighted k-nearest neighbors algorithm and $a_k^T XX^T a_j$ is the weight of each labeled neighbors node j.

Potential Drawbacks. In normal case, this update rule makes sense since if node j is more similar to node k, then node j's label will be added more on node k. However, tricky issues would happen when training of node j is getting better. If prediction of node j's label is trained to be nearly perfect, then node j has impact zero on its neighbors' feature training.

From above lemmas and observations, we can deduce an important theorem:

Theorem 4.1. *If label predictions of a node j almost perfectly fit its ground-truth label during training process, then node j almost has no impact on its neighbors' feature learning.*

This theorem indicates that a well-trained example is "abandoned" during training process, which contradicts to our intuition. This property would probably make GNN fail in some cases (Figure

1). In fact, most of the time we want to fully exploit those well-trained examples to provide more information. One of the intuitive solutions is adaptively assigning more weights to well-trained examples, augmenting their influences to adjacent nodes. In future extensions of this work, we will design new GCN architectures based on this idea and do enough experiments.

Connections to Pre-training Graph Neural Networks. Pre-training graph neural networks is an important topic since this technique is very successful in natural language processing area. Based on our previous analysis, the learning process of two-layer graph neural networks have been divided into two stages: First stage is learning node similarity matrix; Second stage is updating labels based on the node similarity matrix. Graph convolutional networks [8] construct node similarity matrix by comparing k-hop local graph structures between each pair of nodes; Graph attention networks [13] construct node similarity matrix by computing attention coefficients between each pair of nodes; P-GNNs [17] compute each node's relative positions to the selected set of anchor nodes. As a result, the similarity between each pair of nodes would be determined by their distance to anchor nodes. This setting is very similar to our semi-supervised learning settings and we could regard labeled nodes as anchor nodes in P-GNNs. To sum up, pre-training graph neural networks would be equivalent to finding a way of constructing node similarity matrix.

5 Experiment

We do a very simple experiment to record the first appearance of correct predictions of each node, discovering that different nodes have different convergence speed, which may cause potential problems. We run SGC code available in deep graph library github with following settings: learning rate = 0.2, epochs = 100, weight decay = 5e-5 on cora, citeseer and pubmed datasets. We find that most of the nodes get correct predictions after the first epoch and some of them turn to be wrong predictions in the following updates. In short, convergence speed of each node has significant influence on model performance, which agrees with our theoretical results.

6 Conclusion

Through trajectory analysis, we find that the learning process of GNNs could be divided into two stages which are similarity matrix construction and label updates based on similarity matrix. Additionally, we also point out that GNN probably could not efficiently leverage examples to improve training especially "overlook" new well-trained examples. Based on our analysis, this is not only a low-efficiency problem, but will potentially cause naive misclassifications as well. There are many future works we need to do: 1) Generalize trajectory analysis to non-linear cases with different non-linear activation functions; 2) Based on the previous trajectory analysis, we could try to propose new GCN architecture circumventing existed problems; 3) We could also analyze the update rule of unsupervised learning method, like GraphSage algorithm and Graph autoencoders, to see if we could get any other new insights; 4) Pre-training graph to get high-quality initialization of node embeddings for various downstream tasks. We believe that pre-training strategy is highly related to graph kernels and some graph summary statistics.

Acknowledgments

This work is launched in Amazon AI Lab Shanghai. We thank Professor Zheng Zhang, PhD student Wang Chen for helpful discussions in various stages of this work. In addition, we thank members of Amazon AI Lab Shanghai for their guidance on usage of Deep Graph Library, a powerful open-source library for deep learning on graph.

References

[1] S. Du, K. Hou, B. Póczos, R. Salakhutdinov, R. Wang, and K. Xu. Graph Neural Tangent Kernel: fusing graph neural networks with graph kernels. In *NeurIPS*, 2019.

[2] G.E.Hinton, S.Osindero, and Y.-W.Teh. A fast learning algorithm for deep belief nets. In *Neural Computation*, 2006.

[3] W. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[4] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. In *Nature*, 2015.

[5] S. Luan, M. Zhao, X.-W. Chang, and D. Precup. Break the Ceiling: stronger multi-scale deep graph convolutional networks. In *NeurIPS*, 2019.

[6] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. Provably powerful graph networks. In *NeurIPS*, 2019.

[7] C. Morris, M. Ritzert, M. Fey, W. L.Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and Lehman Go Neural: higher-order graph neural networks. In *AAAI*, 2019.

[8] T. N.Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[9] H. NT and T. Maehara. Revisiting Graph Neural Networks: all we have is low-pass filters. In *arxiv https://arxiv.org/abs/1905.09550*, 2019.

[10] M. Schlichtkrull, T. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *ESWC*, 2018.

[11] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. In *Journal of Machine Learning Research*, 2011.

[12] S.V.N.Vishwanathan, N. N.Schraudolph, R. Kondor, and K. M.Borgwardt. Graph kernels. In *Journal of Machine Learning Research*, 2010.

[13] P. Velickovec, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

[14] F. Wu, T. Zhang, A. H. de Souza Jr., C. Fifty, T. Yu, and K. Q.Weinberger. Simplifying graph convolutional networks. In *ICML*, 2019.

[15] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

[16] K. Xu, C. Li, Y. Tian, T. Sonobe, K. ichi Kawarabayashi, and S. Jegelka. Representation learning on graphs with jumping knowledge networks. In *ICML*, 2018.

[17] J. You, R. Ying, and J. Leskovec. Position-aware graph neural networks. In *ICML*, 2019.

[18] X. Zhang and L. Chen. Capsule graph neural network. In *ICLR*, 2019.

Appendix A Proof of Lemma

A.1 Proof of Lemma 3.1

Proof. L = $-\sum_{j\in L} log(softmax((AHW_2)_j)) \cdot y_j$

= $-\sum_{j\in L} log(\frac{e^{(AHW_2)_j} \cdot y_j}{\sum_{t=1}^{c} e^{(AHW_2)_{j,t}}})$

= $-\sum_{j\in L} log(\frac{exp(a_j^T HW_2)\cdot y_j}{\sum_{t=1}^{C} exp((a_j^T HW_2)_t)})$

$\Delta W_2 = \frac{\partial L}{\partial W_2}$ = $-\sum_{j\in L}(\frac{1}{t_3} y_j^T \odot t_4 - \frac{t_4}{t_2})$ = $-\sum_{j\in L} t_0((\frac{y_j^T}{t_3} - \frac{\vec{1}}{t_2}) \odot t_4) = -\sum_{j\in L} H^T a_j \cdot \delta \odot exp(a_j^T HW_2)$

Where $t_0 = H^T a_j$, $t_1 = exp((W_2)^T t_0)$, $t_2 = sum(t_1)$, $t_3 = y_j^T t_1$, $t_4 = exp(a_j^T HW_2)$, $\delta = (\frac{y_j^T}{t_3} - \frac{\vec{1}}{t_2})$

$[AHW_2^{i+1}]_k = [AH(W^i - \Delta W_2)]_k$

= $[AHW_2^i]_k - [AH\Delta W_2]_k$

= $[AHW_2^i]_k + [\sum_{j\in L} AHH^T a_j \cdot \delta \odot exp(a_j^T HW_2^i)]_k$

= $a_k^T HW_2^i + \sum_{j\in L} a_k^T HH^T a_j \cdot \delta \odot exp(a_j^T HW_2^i)$

let's rearrange δ term.

δ = $\frac{(0,...,1_z,...0)}{(0,...,1_z,...0)(exp(W_1^T H^T a_j)_1,...,exp(W_1^T H^T a_j)_C)^T}$ - $(\frac{1}{sum(exp(W_1^T H^T a_j))}, ..., \frac{1}{sum(exp(W_1^T H^T a_j))})$

= $(0, ..., \frac{1}{exp(W_1^T H^T a_j)_z}, ..., 0)$ - $(\frac{1}{sum(exp(W_1^T H^T a_j))}, ..., \frac{1}{sum(exp(W_1^T H^T a_j))})$

= $(-\frac{1}{sum(exp(W_1^T H^T a_j))}, ..., \frac{1}{exp(W_1^T H^T a_j)_z} - \frac{1}{sum(exp(W_1^T H^T a_j))}, ..., -\frac{1}{sum(exp(W_1^T H^T a_j))})$

So $\delta \odot exp(a_j^T HW_1)$

= $(-\frac{1}{sum(exp(W_1^T H^T a_j))}, ..., \frac{1}{exp(W_1^T H^T a_j)_z} - \frac{1}{sum(exp(W_1^T H^T a_j))}, ..., -\frac{1}{sum(exp(W_1^T H^T a_j))}) \odot (exp(a_j^T HW_1)_1, ..., exp(a_j^T HW_1)_C)$

= $(-softmax(a_j^T HW_1)_1, ..., 1 - softmax(a_j^T HW_1)_z, ..., -softmax(a_j^T HW_1)_C)$

= (Ground-truth label of node j - label predictions of node j)

= $y_j^T -$ softmax $(a_j^T HW_2^i) = \epsilon$

So the total update for node k features is:

$\sum_{j\in L} a_k^T XX^T a_j \cdot \epsilon$, where $\epsilon = y_j^T$ - $softmax(a_j^T HW_2^i)$

□

A.2 Proof of Lemma 3.2

Proof. L = $-\sum_{j\in L} log(softmax(A \cdot H \cdot W_2)_j) y_j$, where $H = AXW_1$

L = $-\sum_{j\in L} log(\frac{e^{(AAXW_1W_2)_j} y_j}{\sum_{t=1}^{c} e^{(AAXW_1W_2)_{j,t}}}) = -\sum_{j\in L} log(\frac{exp(a_j^T AXW_1W_2) y_j}{\sum_{t=1}^{C} exp(a_j^T AXW_1W_2)_t)})$

$\frac{\partial L}{\partial W_1}$ = $-\sum_{j\in L} \frac{1}{t_3} \cdot t_0 \cdot ((y_j^T \odot t_4) \cdot W_2^T) - \frac{1}{t_2} \cdot t_0 \cdot (t_4 \cdot W_2^T)$

where $t_0 = X^T \cdot A^T \cdot a_j$, $t_1 = exp(W_1^T \cdot W_2^T \cdot t_0)$, $t_2 = sum(t_1)$, $t_3 = y_j^T \cdot t_1$, $t_4 = exp(a_j^T \cdot A \cdot X \cdot W_1 \cdot W_2)$

$\frac{\partial L}{\partial W_1} = -\sum_{j\in L} t_0(\frac{1}{t_3} \cdot ((y_j^T \odot t_4) \cdot W_1^T) - \frac{1}{t_2} \cdot (t_4 \cdot W_1^T)) = -\sum_{j\in L} X^T A^T a_j \cdot \delta \odot exp(a_j^T \cdot A \cdot X \cdot W_1 \cdot W_2) \cdot W_2^T$

$[A^2 XW_1^{i+1} W_2^i]_k = [A^2 X(W_1^i - \Delta W_1) W_2^i]_k = [A^2 XW_1^i W_2^i]_k$ - $[A^2 X \Delta W_1 W_2^i]_k$

= $a_k^T AXW_1^i W_2^i + \sum_{j\in L} a_k^T AX(AX)^T a_j \cdot \delta \odot exp(a_j^T \cdot A \cdot X \cdot W_1^i \cdot W_2^i) \cdot (W_2^i)^T$

= $a_k^T AXW_1^i W_2^i + \sum_{j\in L} a_k^T AX(AX)^T a_j \cdot \epsilon \cdot (W_2^i)^T$, where $\epsilon = y_j^T$ - $softmax(a_j^T AXW_1^i W_2^i)$ □