Reproducibility Challenge on LazyGNN

ChungYu Wang & Anto Nanah Ji

***Abstract*—The advantages of using deeper graph neural networks (GNNs) to capture long-distance dependency in graphs have been shown in recent publications. However, deeper GNNs face a persistent scalability issue because of large-scale graphs' neighbourhood explosion problem. In LazyGNN[1], Xue et al. proposed to use shallower models rather than deeper models to capture long-distance dependency in graphs. Furthermore, they created the mini-batch LazyGNN and showed that it is compatible with the current scalable approaches such as sampling techniques. They also performed extensive tests and showed that LazyGNN has exceptional scalability and predictive performance on large-scale benchmarks. We plan to reproduce their results. The code is available at https://github.com/wangjohn5507/LazyGNN.**

I. Introduction

A. Problem definition

Recent research in deeper graph neural networks (GNNs) has shown that a performance gain is possible by capturing long-distance relations in graphs. Usually, this is done by stacking more graphs convolution layers or unrolling various fixed point iterations [2]. However, the recursive feature propagation in deeper GNNs causes the well-known neighborhood explosion problem. This is because the number of neighbors grows exponentially with the number of feature propagation layers [3]. This problem leads to enormous scalability challenges for data sampling, computation, memory, parallelism, and end-to-end training with large-scale graphs. In LazyGNN[1], Xue et al. proposed a novel shallow model to capture long-distance dependency in graphs through lazy forward and backward propagation as shown in Figure 1.

B. Dataset

The experiments were conducted in multiple large-scale graph datasets. However, we utilized only the ogbn-arxiv and ogbn-products datasets from the paper which can be readily accessed through the ogb site[1].

C. Proposed solution LazyGNN

In LazyGNN[1], the authors proposed to capture long-distance dependency in graphs by shallower GNNs instead of deeper ones. The key intuition comes from the fact that the computation of feature aggregations is highly correlated and redundant across training iterations because the hidden features in GNNs change slowly. So, we may only need to propagate information lazily by reusing the propagated information over the training iterations. In doing so, the neighbourhood explosion problem is resolved while maintaining the ability to capture long-distance dependency in graphs and dramatically reducing the number of aggregation layers.

[1]https://ogb.stanford.edu/docs/nodeprop

Let $G = (V, E)$ be a graph, where $V = \{v_1, ..., v_N\}$ is the set of nodes and $E = \{e_1, ..., e_M\}$ is the set of edges. Suppose that each node is associated with a d-dimensional feature vector. Let $X_{fea} \in R^{N\times d}$ be the original features for all nodes. The adjacency matrix of G is denoted by $A \in R^{N\times N}$, where $A_{ij}>0$ when there exists an edge between node v_i and v_j , otherwise $A_{i,j} = 0$. We set $\overline{A} = D^{-1/2}AD^{-1/2}$ be the symmetrically normalized graph where D is the degree matrix. Finally, $\overline{L} = I - \overline{A}$ is the **Laplacian** matrix of the symmetrically normalized graph.

Lazy propagation' central concept originates from the most popular and commonly used graph signal denoising problem:

$$min_X \, ||X - X_{in}||_F^2 + (\frac{1}{\alpha} - 1)\, tr(X^T\,(I - \overline{A})X)$$

where the first term maintains the proximity with node hidden features X_{in} after feature transformation, and the second Laplacian smoothing regularization encodes smoothness assumption on graph representations.

Forward Computation. The key insight of LazyGNN is that the approximate solution of the equation (1) evolves smoothly since $(X_{in}^k) = \mathrm{f}(X_{fea},\, \theta^k)$ changes smoothly with model parameters θ^k where k is the index of training iterations. In the equation (2), we mix the diffusion output in iteration $k-1$ into the initial embedding of the diffusion process in training iteration k. In addition, to prevent overfitting, a momentum correction is introduced with hyperparameter β (small β is favored if the dropout rate is small). l in the equation (3) is the index of layers.

$$(X_{in}^k) = \mathrm{f}(X_{fea},\, \theta^k) \tag{1}$$

$$(X_0^k) = (1-\beta)(X_L^{k-1}) + \beta(X_{in}^k) \tag{2}$$

$$(X_{l+1}^k) = (1-\alpha)\overline{A}(X_l^k) + \alpha(X_{in}^k), \forall l = \{0, ..., L-1\} \tag{3}$$

The equation (3) solves the denoising problem of the Graph signal with an implicitly large number of steps. Notice that with a few feature aggregation layers (a small L) we can approximate the fixed point solution of the Graph signal denoising problem.

Key points. Through lazy propagation, LazyGNN is able to capture long-distance dependency in graphs with a small number of feature aggregation layers instead of large number of feature propagation layers, which is less efficient. Furthermore, LazyGNN will not cause the over-smoothing issue instead of many GNN models as the residual connection X_{in}^k in the equation (3) determines the fixed point and prevents the over-smoothed solution.

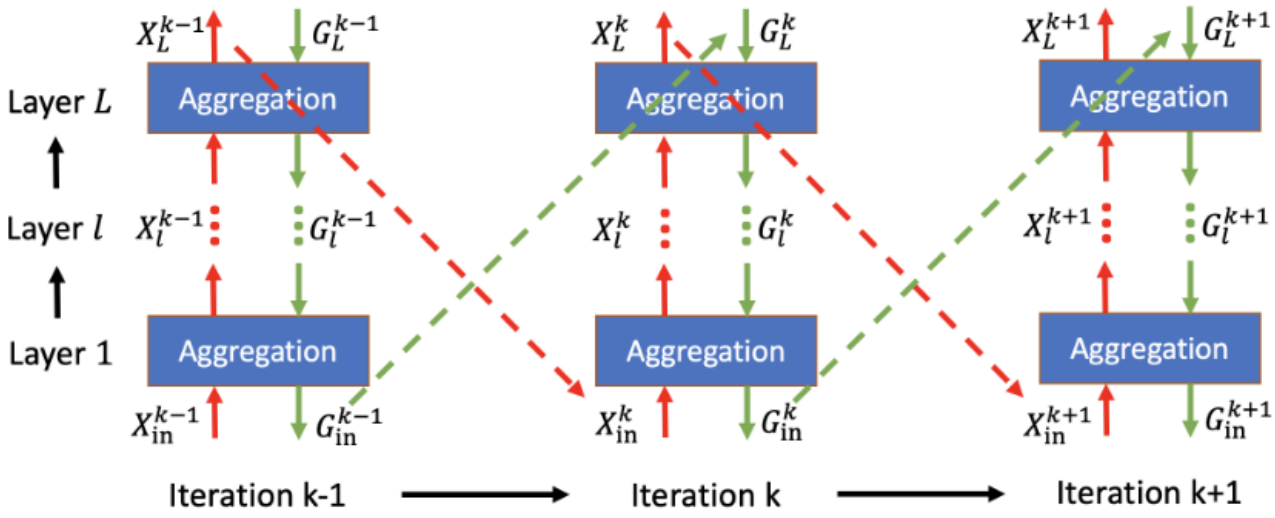

Fig. 1. LazyGNN with Lazy forward (red) and backward (green) propagation.

Backward Computation. it is non-trivial to compute the gradient for the model in the backward computation since the computation graphs from previous training iterations have been destroyed and released in the memory. Thus, in current iterations k, it is not possible to compute the backpropagation through the history variables directly as it is done in general. Consequently, the authors decide to use the implicit function theorem to indirectly compute the gradient.

Theorem 1. *(Implicit Gradient) Let X_* be the fixed point solution of function $g(X_*, X_{in})$, i.e., $g(X_*, X_{in}) = 0$. Given the gradient of loss function $\mathcal{L}(X_*, Y)$ with respect to the fixed point X_*, i.e., $\frac{\partial \mathcal{L}}{\partial X_*}$, the gradient of loss $\mathcal{L}$ with respect to feature X_{in} can be computed as:*

$$\frac{\partial \mathcal{L}}{\partial X_{in}} = -\frac{\partial \mathcal{L}}{\partial X_*}(\mathrm{J}|_{x_*})^{-1}\frac{\partial g(X_*, X_{in})}{\partial X_{in}}$$

where $\mathrm{J}|_{x_*} = \frac{\partial g(X_*, X_{in})}{\partial X_*}$ is the **Jacobian** *matrix of $g(X_*, X_{in})$ evaluated at X_*.*

According to Theorem 1, the gradient of loss $\mathcal{L}$ with respect to X_{in} can be computed as follows:

$$\frac{\partial \mathcal{L}}{\partial X_{in}} = \alpha\frac{\partial \mathcal{L}}{\partial X_*}(I - (1-\alpha)\,\overline{A})^{-1}$$

However, it is still infeasible to compute the expensive matrix inversion. So, they proposed to approximate it by the iterative backward gradient propagation where $\frac{\partial \mathcal{L}}{\partial X_*}$ is approximated by $\frac{\partial \mathcal{L}}{\partial X_L}$, and G_0 provides an approximation for gradient $\frac{\partial \mathcal{L}}{\partial X_{in}}$.

$$G_L = \frac{\partial \mathcal{L}}{\partial X_L}(\approx \frac{\partial \mathcal{L}}{\partial X_*}) \tag{4}$$

$$G_l = \alpha\frac{\partial \mathcal{L}}{\partial X_L} + (1-\alpha)\overline{A}\,G_{l+1}, \forall l = \{L-1, ..., 0\} \tag{5}$$

In addition, they decided to propagate the gradient lazily by leveraging the gradient $\frac{\partial \mathcal{L}}{\partial X_{in}^{k-1}}$ computed in the previous training iteration $k-1$. This is done to reduce the number of gradient propagation layers. Also, to prevent overfitting, a momentum correction is introduced with hyperparameter γ.

$$G_L^k = \gamma\frac{\partial \mathcal{L}}{\partial X_L^k} + (1-\gamma)\frac{\partial L}{\partial X_{in}^{k-1}} \tag{6}$$

$$G_l^k = \alpha\frac{\partial \mathcal{L}}{\partial X_L^k} + (1-\alpha)\overline{A}\,G_{l+1}, \forall l = \{L-1, ..., 0\} \tag{7}$$

Key points. Since LazyGNN uses very few aggregation layers, it significantly reduces the computation and memory cost. Furthermore, LazyGNN provides a fundamental algorithmic improvement to significantly reduce the communication cost in cross-device feature aggregations for distributed GNN training as it uses fewer propagation and communication iterations.

D. Proposed solution for mini-batch LazyGNN

The authors also introduced a mini-batch LazyGNN that enhances the computation and memory efficiency with efficient data sampling as shown in Figure 2. In each training iteration k, they sample a target node set V_1^k and their L-hop neighbor set V_2^k, and they denoted the union of these two nodes as $V^k = V_1^k \cup V_2^k$. An induced subgraph A_{V^k} is constructed based on the node set V^k to run lazy forward and backward propagation.

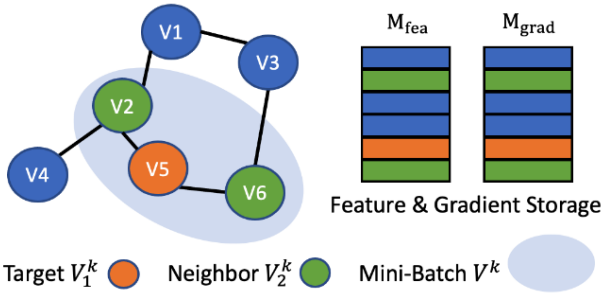

Fig. 2. Mini-batch LazyGNN with Feature & Gradient Storage.

Forward Computation. The node features $(X_{fea})_{V^k}$ for the node set V^k are sampled to form a mini-batch. The corresponding diffused node features $(M_{fea})_{V^k}$ of the same node set are retrieved from the feature storage M_{fea} and then combined with current node features $(X_{in}^k)_{V_k}$.

$$(X_{in}^k)_{V_k} = f((X_{fea})_{V^k}, \theta^k) \tag{8}$$

$$(X_0)_{V_k} = (1-\beta)(M_{fea})_{V^k} + \beta(X_{in}^k)_{V_k} \tag{9}$$

$$(X_{l+1}^k)_{V_k} = (1-\alpha)\overline{A}_{V^k}(X_l^k)_{V^k} + \alpha(X_{in}^k)_{V^k}, \forall l \tag{10}$$

$$(M_{fea})_{V_1^k} = (X_L^k)_{V_1^k} \tag{11}$$

Finally, $(X_L^k)_{V_1^k}$ provides the prediction for target nodes V_1^k, which is further maintained in the features storageM_{fea}. On the other hand, the embedding of neighbor nodes $(X_L^k)_{V_2^k}$ are not accurate and not stored.

Backward Computation. The history gradient $(M_{grad})_{V^k}$ with respect to node features $(X_L^{k-1})_{V^k}$, is retrieved from the gradient storage (M_{grad}) and then combined with $\frac{\partial \mathcal{L}}{\partial (X_L^k)_{V^k}}$, the gradient with respect to current node features $(X_L^k)_{V^k}$.

$$(G_L^k)_{V_k} = (1-\gamma)(M_{grad})_{V^k} + \gamma \frac{\partial \mathcal{L}}{\partial (X_L^k)_{V^k}} \tag{12}$$

$$(G_l^k)_{V_k} = (1-\alpha)\overline{A}_{V^k}(G_{l+1})_{V^k} + \alpha \frac{\partial \mathcal{L}}{\partial (X_L^k)_{V^k}}, \forall l \tag{13}$$

$$(M_{grad})_{V_l^k} = (G_0^k)_{V_l^k} \quad (\approx \frac{\partial \mathcal{L}}{\partial (X_{in}^k)_{V_1^k}}) \tag{14}$$

Finally, $(G_0^k)_{V_l^k}$ provides an estimation for $\frac{\partial L}{\partial (X_{in}^k)_{V_1^k}}$, which is used to compute gradient of the parameters by chain rules and then maintained in the gradient storage (M_{grad}). On the other hand, the gradients of neighbor nodes $\frac{\partial \mathcal{L}}{\partial (X_{in}^k)_{V_2^k}}$ are not accurate and not stored in the gradient memory.

II. Experiments

A. Set up

The original paper does not mention the GPU capacity employed in their experiments; nonetheless, it indicates that the utilization of LazyGNN necessitates merely 2981 MB of GPU memory for the ogbn-products dataset. Based on this data and the effectiveness of LazyGNN, we have chosen to conduct all our local experiments using a GeForce RTX 3060 Lite GPU.

For our research, the dataset was sourced from the OGB organization, given the paper's reliance on ogbn datasets. In our exploration of methods to construct Graph Neural Network (GNN) models tailored to ogbn datasets, we reviewed various projects for insights. This exploration led us to the discovery of the *torch_geometric_autoscale*[4] package[2]. This package is an integral part of the PyTorch Geometric (PyG) ecosystem, specifically designed to enhance the scalability of GNN training and inference for large-scale graphs. It effectively addresses a prevalent issue in GNN deployment: the substantial computational and memory demands for processing extensive graphs often surpass the capabilities of conventional hardware, including GPUs.

By using the *torch_geometric_autoscale* package, we can easily download the ogbn datasets (ogbn-products and ogbn-arxiv) through the following code. `PygNodePropPredDataset(name='ogbn-arxiv')`

B. Reproduce steps

To accurately reproduce the outcomes of the study, we organized our approach into three distinct stages:

- Initially, we developed a straightforward Graph Neural Network (GNN) utilizing the ogbn datasets.
- Subsequently, we applied lazy propagation techniques as described in their study.
- Lastly, we refined our initial GNN model by incorporating lazy propagation into both the forward and backward computational processes.

*1) **Creating GNN model**:* Our initial step involved constructing a basic Graph Neural Network (GNN) modeled after the GCN architecture. Given that the paper did not detail the construction of their model, we designed a basic GNN model featuring two linear layers paired with ReLU activation functions. To prevent overfitting, dropout layers were introduced following each ReLU layer. The number of layers in the model, dropout rate, peak learning rate, and the number of hidden channels were treated as adjustable hyperparameters. To train the model effectively and to ensure smooth convergence, we opted for the **Adam** optimizer coupled with a **cosine** warmup learning rate scheduler.

Additionally, we performed a grid search to identify the optimal set of hyperparameters, mirroring the methodology employed in the paper. Finally, we started to train this model using the ogbn-arxiv and ogbn-products datasets. This approach was undertaken to confirm our comprehension of GNN implementation and to ensure the datasets were accurately acquired.

*2) **Implementing lazy propagation**:* According to the paper, lazy propagation is designed to capture long-distance dependencies in graphs efficiently by reusing computed features

[2]https://github.com/rusty1s/pyg_autoscale

and gradients over training iterations. This approach aims to address the scalability challenges associated with deep GNNs, particularly the neighborhood explosion problem. Therefore, we first start with the forward pass of lazy propagation.

Forward pass. The forward method takes several inputs, including the node features `x`, the adjacency matrix `adj_matrix`, a unique identifier `id`, the number of target nodes `size`, the number of propagation steps `K`, coefficients `alpha` and `beta` for the mixing of the adjacency matrix and node features, and `theta` for the mixing of current and historical gradients. It also takes `equ_preds` and `equ_grad` for storing node features and gradients from previous iterations, respectively, and the computation `device`.

After having inputs for the function, we initialize the tensors `feature` and `grad` to store the node features and gradients from previous iterations. These tensors are then updated with values from `equ_preds` and `equ_grad` based on the provided `id`.

For the lazy forward propagation, we check if the `feature` tensor for the target nodes is uninitialized (i.e., all zeros). If uninitialized, it uses the current node features `x`. Otherwise, it mixes the historical features with the current features using the coefficient `beta`. This step effectively reuses computations from previous iterations, which is the essence of lazy propagation.

The paper uses feature aggregation to approximately compute the historical feature matrix. Therefore, for the feature aggregation, we perform graph convolutional aggregation over `K` propagation steps. In each step, it mixes the transformed node features with the original features using the coefficient `alpha`. This step spreads information across the graph, capturing long-distance dependencies.

Backward pass. For the backward pass, we first compute gradients with respect to the node features. It similarly utilizes lazy propagation for the gradients, mixing the historical gradients `grad` with the current gradients `grad_output` from the loss function using the coefficient `theta`.

Then, we propagate the gradients through the graph for `K` steps, similar to the forward pass. This propagation ensures that the computed gradients reflect the influence of long-distance node dependencies.

While the original paper does not highlight the use of gradient masking to target particular nodes, our review of additional literature indicates the importance of incorporating gradient masking. This is because gradient masking is a widespread technique that may not be specifically mentioned in every paper related to Graph Neural Networks (GNNs) unless it is a key aspect of the proposed approach or essential for a certain model architecture or application. For implementing gradient masking, the gradients for non-target nodes (nodes beyond `size`) are set to zero to ensure that only the gradients of the target nodes are updated.

*3) **Adjust original GNN model**:* Following the development of lazy propagation, we modified our model to incorporate this technique during the training phase. This approach leverages intermediate node representations and gradient information across training iterations, optimizing performance and computational efficiency for large-scale graphs.

To integrate lazy propagation into the original model, a backward hook is registered to the output of the final linear transformation. This hook captures the gradients as they are propagated back through the network during the backward pass. Within this hook, gradients are stored in `self.grad_memory`, specifically for the target nodes identified by `id[:size]`. This selective storing is part of the lazy propagation strategy, where only relevant gradients are kept for future iterations, enhancing memory efficiency and focusing learning on significant nodes.

The `prop1.apply` method is then called with the transformed features and other parameters necessary for lazy propagation. This function encapsulates the logic for the customized propagation as described in the lazy propagation. It utilizes both the adjacency matrix `adj` and parameters like `alpha`, `beta`, and `theta` to perform a mix of current and historical node features and their gradients in a manner optimized for graph-structured data.

Post lazy propagation, the features for the target nodes are updated in `self.feature_memory`. This ensures that the most recent node features are stored after being processed, which will be reused in subsequent training iterations. This step is crucial for implementing the lazy propagation strategy, where past computations are leveraged to reduce future computation needs.

Finally, the output `z_out` from the lazy propagation step is passed through a log softmax function to normalize the outputs, which is typically used in multi-class classification tasks to obtain probabilities.

III. Result

To assess our replicated model and compare it with the original study, we undertook an ablation study focusing on key variables such as the number of layers, batch size, and various hyperparameters. Additionally, we include a figure demonstrating the convergence of LazyGNN, as the original paper illustrated that LazyGNN achieves quicker convergence.

Ablation study on number of layers. Drawing inspiration from Table 4 in the original paper, we performed an ablation study to examine the effect of varying the number of layers in our model. Specifically, we evaluated our model with two, three, and four layers, respectively, using the ogbn-arxiv and ogbn-products datasets. The results are shown in table I.

Observations reveal that the optimal results for the ogbn-arxiv dataset were achieved with a four-layer configuration, with performance diminishing marginally in models featuring fewer layers. This outcome aligns with the original paper's findings for the ogbn-arxiv dataset, albeit with a slight discrepancy in accuracy where the original paper's model outperforms

TABLE I
ABLATION STUDY ON NUMBER OF LAYERS (2, 3, 4). SHOWING THE RESULT OF ACCURACY (%) ON VALIDATION DATA WITH DIFFERENT SETTINGS.

Methods	OGBN-ARXIV (%)	OGBN-PRODUCTS (%)
LazyGNN (L=2)	71.26%	Out of Memory
LazyGNN (L=3)	71.66%	Out of Memory
LazyGNN (L=4)	**71.78%**	Out of Memory

TABLE II
ABLATION STUDY ON DIFFERENT BATCH SIZE (50, 100, 150, 200). SHOWING THE RESULT OF ACCURACY (%) ON VALIDATION DATA WITH DIFFERENT SETTINGS.

Batch Size	50	100	150	200
Accuracy (%)	**72.54%**	72.48%	72.32%	72.20%

ours. In the case of the ogbn-products dataset, we faced challenges in replicating the results due to GPU memory constraints, experiencing out-of-memory issues even when testing the model with only two layers.

Ablation study on batch size. Drawing inspiration from Table 5 in the original paper, we performed an ablation study to examine the effect of different batch sizes on our model. Specifically, we evaluated our model with 50, 100, 150, and 200 batch sizes, respectively, using the ogbn-arxiv datasets. The results are shown in table II.

Observations reveal that the optimal results for the ogbn-arxiv dataset were achieved when the batch size is 50. This outcome aligns with the original paper's findings for the ogbn-arxiv dataset, even though with a slight difference in accuracy where the original paper's model outperforms ours. Similarly, in the case of the ogbn-products dataset, we faced challenges in replicating the results due to GPU memory constraints. At the end, we can conclude that a smaller batch size brings better performance because the sampling variance is reduced.

Ablation study on hyperparameters. Drawing inspiration from Table 6 in the original paper, we performed an ablation study to examine the effect of varying hyperparameters β and γ. Specifically, we evaluated our model with 0.0, 0.5, 0.9, and 1.0 values for β and γ, using the ogbn-arxiv and ogbn-products datasets. The results are shown in table III.

Observations reveal that the optimal results for the ogbn-arxiv dataset were achieved when setting $\beta = 0.5$ and $\gamma = 0.5$ because it achieves a good trade-off between historical information and current iteration. Setting $\beta = 0$ and $\gamma = 0$ produces the worst performance because it fully trusts the history embedding that might be outdated. Setting $\beta = 1$ and $\gamma = 1$ performs slightly better. These outcomes align with the original paper's findings for the ogbn-arxiv dataset, with a slight discrepancy in accuracy. Similarly, we faced challenges in replicating the results with ogbn-products dataset due to GPU memory constraints.

Convergence. Figure 3 illustrates the training process convergence of LazyGNN using the ogbn-arxiv dataset. The validation accuracy convergence depicted in this figure indicates that

TABLE III
ABLATION STUDY ON DIFFERENT COMBINATION OF HYPERPARAMETERS (β, γ). SHOWING THE RESULT OF ACCURACY (%) ON THE AGBN-ARXIV DATASET.

Hyperparameter settings	Accuracy (%)
β=0.0, γ=0.0	69.94%
β=1.0, γ=1.0	71.85%
β=0.5, γ=0.1	72.28%
β=0.5, γ=0.5	**72.95%**
β=0.5, γ=0.9	72.21%
β=0.1, γ=0.5	72.22%
β=0.5, γ=0.5	**72.95%**
β=0.9, γ=0.5	72.34%

LazyGNN achieves a similar rate of convergence with respect to the number of training epochs.

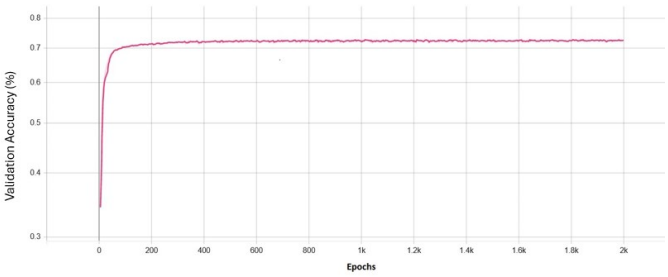

Fig. 3. Accuracy on validation data.

IV. RELATED WORK

It is well known that it is beneficial to capture long-distance relations in graphs. Usually, this is done by stacking more feature aggregation layers or unrolling various fixed point iterations in GNNs [2]. But these works suffer from scalability concerns due to the neighborhood explosion problem [3].

Too many existing research focuses on improving the efficiency and scalability of large-scale GNNs using different designs such as sampling methods, pre-computing or post-computing methods, and distributed methods.

- The Sampling method, where nodes and edge are sampled, mitigates the neighbor explosion problem by either removing neighbors or updating with feature memory [3].
- The strategy of the pre-computing method is to pre-compute the feature aggregation before training [5].
- On the other hand, post-computing method post-process feature aggregation with label propagation after training [6].
- Distributed methods distribute large graphs to multiple servers and parallelized GNNs training [7].

In LazyGNN[1], Xue et al. proposed different approach to capture long-distance dependency in graphs by shallower models and applying lazy forward and backward propagation as shown in Figure 1.

V. Discussion

We were able to replicate the results of the original study, affirming the validity of our experimental approach. Despite minor discrepancies in performance metrics compared to those documented in the original paper, our findings corroborate the paper's conclusions, demonstrating that our model faithfully mirrors the original design. Nonetheless, our efforts to reproduce the results on the ogbn-products dataset were hindered by constraints related to GPU memory capacity. This limitation underscores the necessity for optimizing computational resources or seeking alternative strategies to manage larger datasets effectively.

VI. Contribution of each team member

Together, we collaborated on constructing the LazyGNN model using PyTorch. Chung-Yu was responsible for establishing the experimental framework and carrying out experiments to compare performance. Meanwhile, Anto conducted the ablation studies to assess the influence of hyperparameters and was in charge of documenting and reporting the findings from these experiments.

References

[1] Rui Xue, Haoyu Han, MohamadAli Torkamani, Jian Pei, and Xiaorui Liu. Lazygnn: Large-scale graph neural networks via lazy propagation, 2023.

[2] Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank, 2018.

[3] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[4] M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec. GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning (ICML)*, 2021.

[5] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr. au2, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks, 2019.

[6] Chuxiong Sun, Hongming Gu, and Jie Hu. Scalable and adaptive graph neural networks with self-label-enhanced training, 2021.

[7] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 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 amp; Data Mining*, KDD '19. ACM, July 2019.