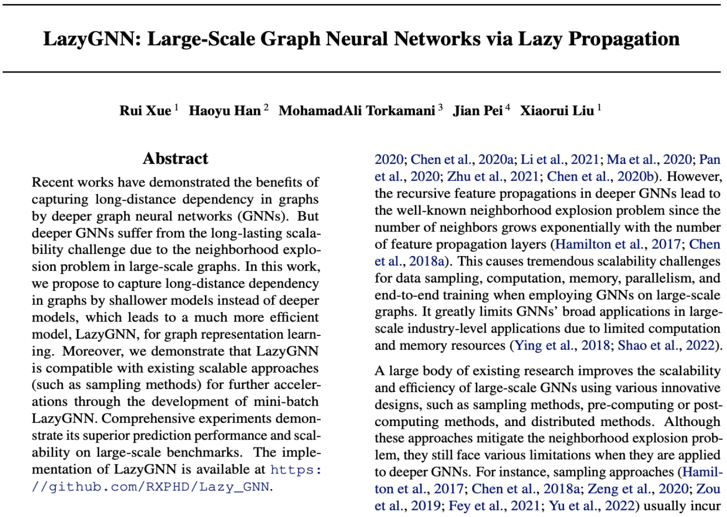
Lazy Graph Neural Networks

ChungYu Wang & Anto Nanah Ji

Paper



LazyGNN: Large-Scale Graph Neural Networks via Lazy Propagation

Rui Xue [1] Haoyu Han [2] MohamadAli Torkamani [3] Jian Pei [4] Xiaorui Liu [1]

Abstract

Recent works have demonstrated the benefits of capturing long-distance dependency in graphs by deeper graph neural networks (GNNs). But deeper GNNs suffer from the long-lasting scalability challenge due to the neighborhood explosion problem in large-scale graphs. In this work, we propose to capture long-distance dependency in graphs by shallower models instead of deeper models, which leads to a much more efficient model, LazyGNN, for graph representation learning. Moreover, we demonstrate that LazyGNN is compatible with existing scalable approaches (such as sampling methods) for further accelerations through the development of mini-batch LazyGNN. Comprehensive experiments demonstrate its superior prediction performance and scalability on large-scale benchmarks. The implementation of LazyGNN is available at `https://github.com/RXPHD/Lazy_GNN`.

2020; Chen et al., 2020a; Li et al., 2021; Ma et al., 2020; Pan et al., 2020; Zhu et al., 2021; Chen et al., 2020b). However, the recursive feature propagations in deeper GNNs lead to the well-known neighborhood explosion problem since the number of neighbors grows exponentially with the number of feature propagation layers (Hamilton et al., 2017; Chen et al., 2018a). This causes tremendous scalability challenges for data sampling, computation, memory, parallelism, and end-to-end training when employing GNNs on large-scale graphs. It greatly limits GNNs' broad applications in large-scale industry-level applications due to limited computation and memory resources (Ying et al., 2018; Shao et al., 2022).

A large body of existing research improves the scalability and efficiency of large-scale GNNs using various innovative designs, such as sampling methods, pre-computing or post-computing methods, and distributed methods. Although these approaches mitigate the neighborhood explosion problem, they still face various limitations when they are applied to deeper GNNs. For instance, sampling approaches (Hamilton et al., 2017; Chen et al., 2018a; Zeng et al., 2020; Zou et al., 2019; Fey et al., 2021; Yu et al., 2022) usually incur

Outline

Problem definition

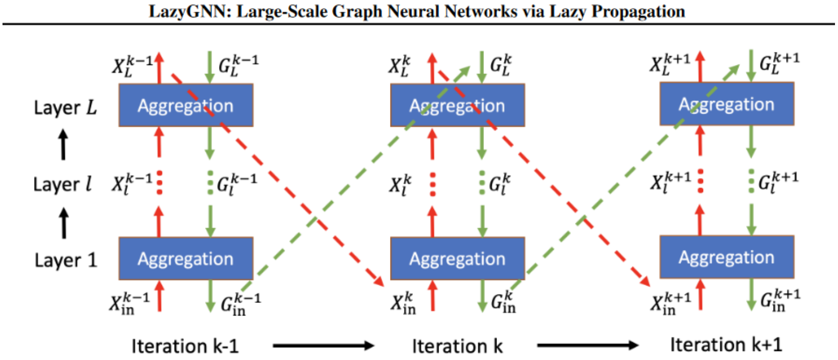
- Capturing long-distance dependency in graphs has a lot of benefits.
- However, GNNs suffer from neighborhood explosion problem.
- This is because the number of neighbors grows exponentially with the number of feature propagation layers.
- This problem leads to enormous scalability challenges for data sampling, computation, and memory.

Problem definition

There are known ways to capture long-distance relations in graphs:

- Stacking more graph convolution layers.
- Unrolling various fixed point iterations.

Proposed solution



Use shallower GNNs and apply lazy forward & backward propagation.

Datasets

Dataset	#Nodes	#Edges
ogbn-arxiv	169,343	1,166,243

Forward computation

Notations:

- 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.
- Let $A \in R^{N \times N}$ be the adjacency matrix of G, where $A_{ij} > 0$ when there exists an edge between node v_i and v_j , otherwise $A_{i,j} = 0$.
- Let $\overline{A} = D^{-1/2} A D^{-1/2}$ be the symmetrically normalized graph where D is the degree matrix.
- Let $\overline{L} = I - \overline{A}$ be the **Laplacian** matrix of the symmetrically normalized graph.

Forward computation

Graph signal denoising problem:

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

Forward computation

Feature transformation

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

where

- k - Training iterations
- f - Feature transformation function
- θ - Model parameters

Forward computation

Lazy propagation

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

where β - Hyperparameter

Forward computation

Lazy propagation

```
### forward lazy propagation & momentum connection

if torch.equal(feature[:self.size],
↪  torch.zeros_like(feature)[:self.size].to(self.device)):
    f = x
else:
    f = (1-self.beta)*feature + self.beta*x #target
     ↪  nodes
```

Forward computation

Feature layers aggregation

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

Forward computation

Feature layers aggregation

```
### aggragation

for i in range(self.K):
    f = (1 - self.alpha) * (self.adj_matrix @ f) +
     ↪  self.alpha * x
return f
```

Forward computation

$$(X_{in}^{k}) = \mathrm{f}(X_{fea},\, \theta^{k})$$
$$(X_{0}^{k}) = (1-\beta)(X_{L}^{k-1}) + \beta(X_{in}^{k})$$
$$(X_{l+1}^{k}) = (1-\alpha)\overline{A}(X_{l}^{k}) + \alpha(X_{in}^{k}), \forall l = \{0, ..., L-1\}$$

Remarks:

1. Long-distance dependency.
2. Over-smoothing.

Backward Computation

Theorem: (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:

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

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

Backward Computation

The gradient of loss $\mathcal{L}$ with respect to X_{in} according to implicit gradient theorem

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

Backward Computation

The approximation of the expensive matrix inversion

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

Backward Computation

The approximation of the expensive matrix inversion

```
for j in range(self.K):
    g = (1 - self.alpha) * (self.adj_matrix @ g) +
     ↪  self.alpha * grad_output
```

Backward Computation

Lazy propagation of the gradient

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

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

where γ - Hyperparameter

Backward Computation

```
###backward lazy propagation & momentum connection

if torch.equal(grad[:self.size],
↪  torch.zeros_like(grad[:self.size].to(self.device)):
    g = grad_output
else:
    g = (1-self.theta)*grad + self.theta*grad_output
```

Backward Computation

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

Remarks:

1. Computation and memory efficiency.
2. Communication efficiency.

model

```
class Net(torch.nn.Module):
    def __init__(self, num_features, hidden_channels,
     ↪  num_classes, num_layers, num_nodes, dropout,
     ↪  **kwargs):
        super(Net, self).__init__()
        self.linears = torch.nn.ModuleList()

         ↪  self.linears.append(torch.nn.Linear(num_features,
         ↪  hidden_channels))
        for _ in range(num_layers - 2):

             ↪  self.linears.append(torch.nn.Linear(hidden_cha
             ↪  hidden_channels))


         ↪  self.linears.append(torch.nn.Linear(hidden_channel
         ↪  num_classes))
        self.prop1 = Lazy_Prop()
```

Experiment results

Ablation study:

- Lazy propagation
- Batch size
- Sensitivity analysis on hyperparameters β and γ

Lazy propagation experiment results

#Linear layers	Accuracy(%)
2	71.26%
3	71.66%
4	71.78%

Batch size experiment results

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

Sensitivity analysis experiment results

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%

Convergence

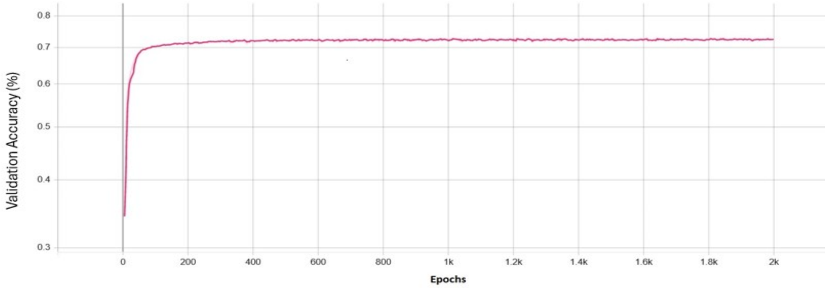


Figure: Accuracy on validation data

Conclusions

- Fortunately, we were able to reproduce their results.
- Overall the performance measures that we have is slightly different that the paper.

Thank you!!