PinSage++ *

Siyang Qu
siqyangq@andrew.cmu.edu

Yue Wu
ywu5@andrew.cmu.edu

Abstract

Recent advancements in deep neural networks for graph-structured data have led to state-of-the-art performance on recommender system benchmarks. However, making these methods practical and scalable to web-scale recommendation tasks with billions of items and hundreds of millions of users remains a challenge.

In this work, we present a web-scale recommender system based on graph convolutional network (GCN). Since the graph does not fit in a single GPU, we distribute the graph across multiple NVIDIA GPUs with METIS. Each GPU worker pushes gradient to the parameter server which has access to the whole graph. We primarily experimented with two version of implementations: A baseline distributed GraphSAGE model, and an optimized PinSAGE model with Producer-Consumer computation graph generation and Map-Reduce Inference. Our optimized model achieves significant performances improvement over the the baseline, and the best sequential implementation.

1 Summary

We parallelized training and inference of a Graph Convolution Network (GCN) on 2 GPUs and 32 CPUs to achieve a 12x speedup during training and 10x speedup during inference.

2 Introduction

Deep learning methods have an increasingly critical role in recommender system applications, being used to learn useful low dimensional embeddings of images, text, and even individual users. The representations learned using deep models can be used to complement, or even replace, traditional recommendation algorithms like collaborative filtering. and these learned representations have high utility because they can be re-used in various recommendation tasks. For example, item embeddings learned using a deep model can be used for item-item recommendation and also to recommended themed collections (e.g., playlists, or "feed" content).

Recent years have seen significant developments in this space, especially the development of new deep learning methods that are capable of learning on graph-structured data, which is fundamental for recommendation applications (e.g., to exploit user-to-item interaction graphs as well as social graphs)

Most prominent among these recent advancements is the success of deep learning architectures known as Graph Convolutional Networks (GCNs)

The most computationally expensive operation is the application of the convolve step to a batch of networks. The inefficiency comes from the fact that each GPU process needs to access neighborhoods and feature information of nodes in the k-hop neighborhood that are stored in CPU memory. Training can thus benefit from parallelizing neighborhood sampling and subgraph construction on multiple CPUs, partitioning the original graph and parallelizing training on multiple GPUs with a parameter

*Project homepage: qusiyang.github.io

server. This work distribution between CPU and GPU allows CPU processes to precompute subgraph adjacency list and neighborhood's feature data such that it can be passed together with the minibatch training data to GPU, minimizing CPU and GPU communication.

3 Background

The main challenge of our project is to scale both the training as well as inference of GCN-based node embeddings to graphs with billions of nodes and tens of billions of edges. Scaling up GCNs is difficult because many of the core assumptions underlying their design are violated when working in a big data environment. For example, most existing GCN-based recommender systems require operating on the full graph Laplacian during training—an assumption that is infeasible when the underlying graph has billions of nodes and whose structure is constantly evolving.

Hamilton et al. [2017] proposed to sample the neighborhood with random walk instead of calculating the full laplacian. This greatly reduced the amount of computation, and made GCNs feasible for larger datasets the first time. However, the proposed solution only enables researchers to work on larger toy datasets, since the whole graph has to fit into the memory of a single GPU. In this project, we will build upon their code as it is the best publically available baseline in the field.

Ying et al. [2018] further extends Hamilton et al. [2017]'s work to a multiple-gpu/worker implementation with a producer-consumer minibatch construction strategy. A large-memory, CPU-bound producer efficiently samples node network neighborhoods and fetches the necessary features to define local convolutions, while a GPU-bound TensorFlow model consumes these pre-defined computation graphs to efficiently run stochastic gradient decent. The biggest problem is that there is not code publicly available, and **most implementation details have been omitted** due to the business value of the proposed method.

We further identify the following problems:

1. The CPU workers push all the built computation graphs to the same location despite they are independent and will be "consumed" by different GPU workers. This is a potential waste resources when the problem is shared across different machines.

2. The mapReduce inference framework provided by the paper applies only to bipartite graph. Inference on non-bipartite graphs requires repeated computation of node embeddings caused by the overlap between k-hop neighborhoods of nodes shown in Figure 1.

The key data structure that we are dealing with is a graph with approximately 240000 nodes and 12000000 edges from the reddit dataset. Key operations on this data structure involve partitioning and neighbor sampling for training and independent set finding for the mapReduce inference framework.

With the above problems, the goal of the algorithm is to speedup training and inference of a general graph without degrading the network's prediction accuracy on common classification tasks. Given a graph where nodes can represent users or items and edges represent some form of relationship, the goal of training is to learn a function that aggregates information from the k-hop neighbors of each node. With the reddit dataset, we operate on a graph with approximately 240000 nodes and 12000000 edges and key operations involve graph partitioning and neighborhood sampling. The outputs of training is the network model where learned weights specify how we combine neighborhood node embeddings to compute embedding for a given node. The goal of inference is to then apply the trained model to generate embedding for all items, including those that were not seen during training. Directly applying an iteration of the training algorithm is inefficient due to repeated computations and we employ a greedy approach to find independent sets in graph so that the PinSage's mapReduce framework on bipartite graphs can be modified and applied to a general graph. These node embeddings can then be used for downstream tasks such as classification or recommendation via nearest-neighbor lookups.

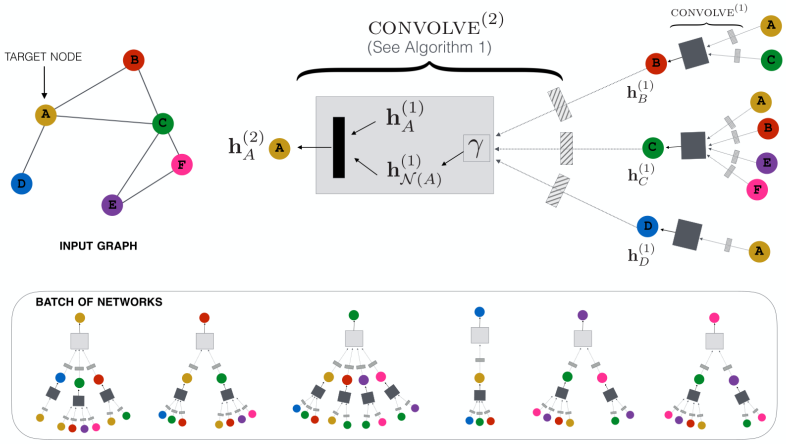

Figure 1: Overview of our model architecture using depth-2 convolutions (best viewed in color). Left: A small example input graph. Right: The 2-layer neural network that computes the embedding $h_A^{(2)}$ of node A using the previous-layer representation, $h_A^{(1)}$, of node A and that of its neighborhood $\mathcal{N}(A)$ (nodes B, C, D). (However, the notion of neighborhood is general and not all neighbors need to be included (Section 3.2).) Bottom: The neural networks that compute embeddings of each node of the input graph. While neural networks differ from node to node they all share the same set of parameters (*i.e.*, the parameters of the CONVOLVE$^{(1)}$ and CONVOLVE$^{(2)}$ functions; Algorithm 1). Boxes with the same shading patterns share parameters; γ denotes an importance pooling function; and thin rectangular boxes denote densely-connected multi-layer neural networks.

4 Methods

4.1 Model Architecture

The PinSage model uses localized convolution modules to generate embeddings. Algorithm 1 details the CONVOLVE step that specifies a set of parameters Q,W,q,w that aggregate neighborhood information.

Algorithm 1: CONVOLVE

Input : Current embedding $\mathbf{z}_u$ for node u; set of neighbor embeddings $\{\mathbf{z}_v | v \in \mathcal{N}(u)\}$, set of neighbor weights $\boldsymbol{\alpha}$; symmetric vector function $\gamma(\cdot)$

Output : New embedding $\mathbf{z}_u^{\text{NEW}}$ for node u

$\mathbf{n}_u \leftarrow \gamma\left(\{\text{ReLU}\left(\mathbf{Q}\mathbf{h}_v + \mathbf{q}\right) \mid v \in \mathcal{N}(u)\}, \boldsymbol{\alpha}\right)$;

$\mathbf{z}_u^{\text{NEW}} \leftarrow \text{ReLU}\left(\mathbf{W} \cdot \text{CONCAT}(\mathbf{z}_u, \mathbf{n}_u) + \mathbf{w}\right)$;

$\mathbf{z}_u^{\text{NEW}} \leftarrow \mathbf{z}_u^{\text{NEW}} / \|\mathbf{z}_u^{\text{NEW}}\|_2$

Algorithm 2: MINIBATCH

Input : Set of nodes $\mathcal{M} \subset \mathcal{V}$; depth parameter K; neighborhood function $\mathcal{N} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$

Output : Embeddings $\mathbf{z}_u, \forall u \in \mathcal{M}$

/* Sampling neighborhoods of minibatch nodes. */

$\mathcal{S}^{(K)} \leftarrow \mathcal{M}$;

for $k = K, ..., 1$ **do**

  $\mathcal{S}^{(k-1)} \leftarrow \mathcal{S}^{(k)}$;

  for $u \in S^{(k)}$ **do**

    $\mathcal{S}^{(k-1)} \leftarrow \mathcal{S}^{(k-1)} \cup \mathcal{N}(u)$;

  end

end

/* Generating embeddings */

$\mathbf{h}_u^{(0)} \leftarrow \mathbf{x}_u, \forall u \in \mathcal{S}^{(0)}$;

for $k = 1, ..., K$ **do**

  for $u \in \mathcal{S}^{(k)}$ **do**

    $\mathcal{H} \leftarrow \left\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u)\right\}$;

    $\mathbf{h}_u^{(k)} \leftarrow \text{CONVOLVE}^{(k)}\left(\mathbf{h}_u^{(k-1)}, \mathcal{H}\right)$

  end

end

for $u \in \mathcal{M}$ **do**

  $\mathbf{z}_u \leftarrow \mathbf{G}_2 \cdot \text{ReLU}\left(\mathbf{G}_1 \mathbf{h}_u^{(K)} + \mathbf{g}\right)$

end

To gain more information about the local graph structure around a node, we use multiple layers of convolutions that corresponds to k-hop neighborhoods where each layer requires the output of

previous layer as input. The output of the final layer is used to generate the final node embedding. The detail of this algorithm is shown in Algorithm 2.

4.2 Parameter Server

We train the PinSAGE model with a single parameter server and a bunch of CPU-GPU worker threads. Each worker obtains a local copy of the network, and compute the gradient with respect to a local sample from the training set. The central parameter server then accumulates the gradients and update the weight of the global network.

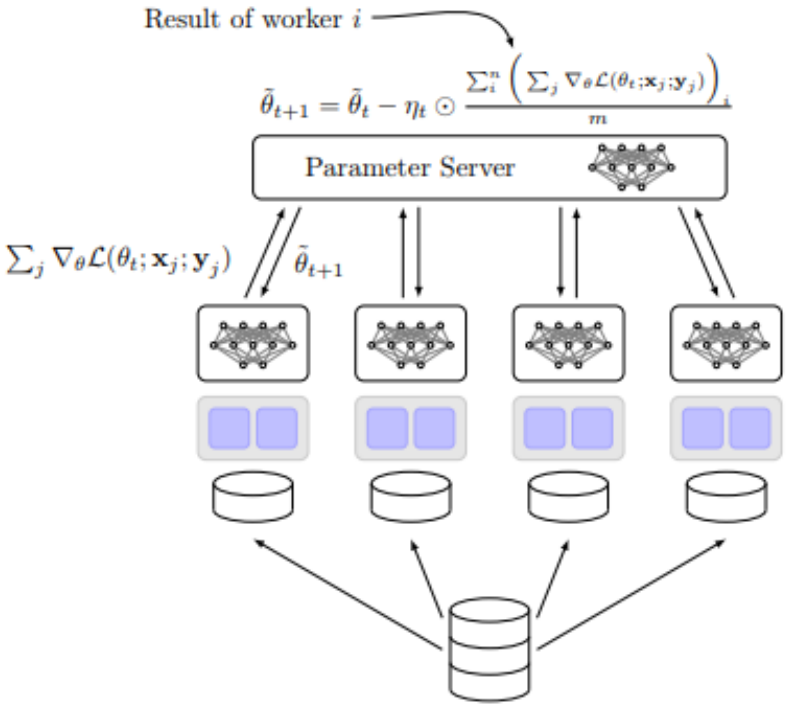

Figure 2: Illustration of a parameter server.

4.3 Producer-Consumer Computation Graph Generation

While the parameter server framework allowed for distributed machine learning training, we observed that both GPU and CPU usage were low, as the majority of time was spent on synchronizing and waiting between CPU-bound random-walk neighborhood sampling (Computation Graph Generation) and GPU-bound training (Computation Graph Evaluation).

We develop a producer-consumer architecture for constructing minibatches that promotes maximal GPU utilization during model training. A large-memory, CPU-bound producer efficiently samples node network neighborhoods and fetches the necessary features to define local convolutions, while a GPU-bound Pytorch model consumes these pre-defined computation graphs to efficiently run stochastic gradient decent.

4.4 Map-Reduce Inference

After the model is trained, we are interested in generating node embeddings for all nodes in the graph at inference time. The obvious approach is to use one iteration of model training where we go through minibatches of nodes, perform sampling, and apply Algorithm 2. This approach, however, leads to repeated computations caused by overlapping k-hop neighbors between nodes that are near each other. As seen in Figure 1, it is common for the same node to exist in different networks used to

generate embeddings for different target nodes.

PinSage addresses the issue through a MapReduce pipeline that runs model inference without repeated computations. The MapReduce job has two parts. First, project all pins to a low-dimension latent space (Algorithm 1, Line 1). Second, compute board embeddings by pooling features of its neighbors (all pins). This process is repeated k times for both pins and boards where k is the depth of convolution which we defined in training. This works as the pin-to-board Pinterest graph is bipartite by construction, and by experimentation, this process is a good enough approximation for a training iteration.

This approach, however, does not work with the reddit dataset or generic non-bipartite graphs that we come across in many real world applications. Our approach is to greedily partition the graph into independent sets of nodes during precomputation. The MapReduce method can then be applied to k disjoint sets of vertices (k is the number of independent sets) instead of two in the bipartite case. Experiments show significant speedup while over baseline while maintaining classification accuracy.

5 Experiments

5.1 Reddit Dataset

Reddit is an online discussion forum where users post and comment on content under different topics. A graph dataset is constructed where each node represents a post and the node label is the community, or "subreddit", that the post belongs to. 50 large communities were sampled and we connect two posts (nodes) if the same user comments on both. The graph contains a total of 232,965 posts with an average degree of 492. Each node is also associated with a 300-dimensional GloVe (Pennington et al. [2014]) embedding of the corresponding post.

5.2 Parameter Server Basline Training

Our baseline implementation is a single core algorithm that simply loops through minibatches of training data and performs algorithm 2. The first improvement over the simple baseline was the implementation of a synchronous parameter server. As shown in Figure 2, the parameter server holds a copy of the model and during training, it receives gradients from workers and apply them to the model and then send the updated gradients back to each worker. We use two worker nodes, each running on a single GPU, drawing training samples from it's local graph partition. Each worker receives a partition of the training data, and runs independently to compute local subgradients.

5.3 Producer-Consumer

In addition to the GPU workers and Parameter Server as described above. We isolate random-walk sampling from the GPU workers and assign a number of CPU workers to traverse the graph.

We experimented with different number of CPU workers to confirm our hypothesis.

5.4 Inference

We measure the baseline performance of inference via running a single iteration of the single core algorithm. We experimented with different batch sizes, and as shown in table 2, we achieved a 10x speedup compared with the baseline model with a batch size of 128. Note that further increasing the batch size negatively affects our loss during training.

6 Results

6.1 Speedup over Baseline

We evaluate our improved implementation against baseline and achieved at least 6x speedup. From the significant speedup in Figure 3, we confirm that the training process of PinSAGE can greatly benefit from the producer-consumer training architecture.

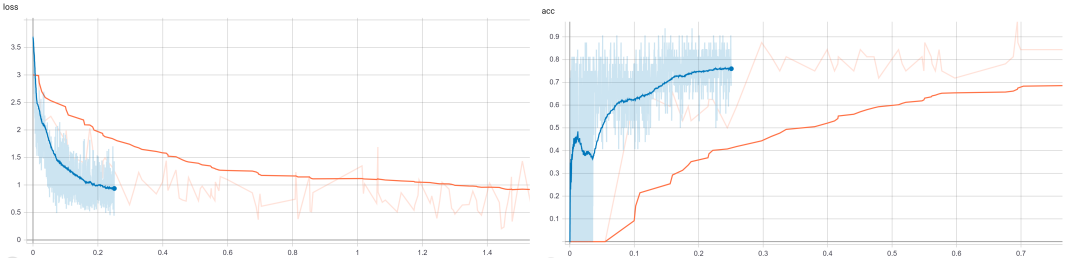

Figure 3: Comparison between the baseline (orange) and Producer-Consumer with 6 CPU workers (blue). The model reaches target accuracy (vertical axis) in approximately 1/6 time (horizontal axis). Training loss is shown on the left, and accuracy is shown on the right.

6.2 Speedup with respect to number of CPU workers

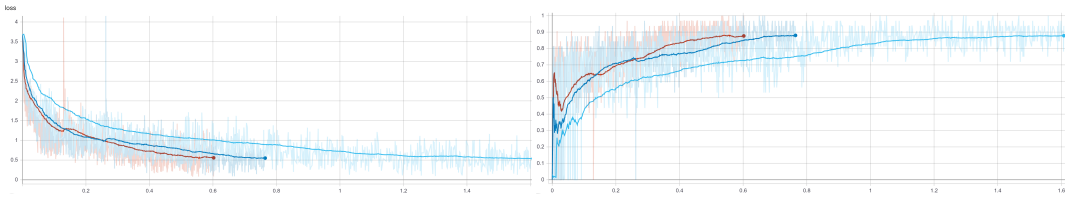

Figure 4: Comparison of the speedup gained by increasing the number of CPU workers. The number of CPU workers tested are "2:teal, 6:blue, 12:red" Training loss is shown on the left, and accuracy is shown on the right.

Table 1: Analysis of the average wait-time of GPU/CPU workers with respect to the number of CPU workers. Wait-time refers to the time the whole system have to wait for job to proceed. For example, GPU wait-time refers to the time the whole system has to wait to assign a job to an available GPU worker.

# CPU Workers	GPU Job Wait Time (s)	CPU Job Wait Time (s)
12	0.038	0.406
8	0.076	0.557
4	0.028	0.673
2	0.00012	1.14

From 2 and 4, we deduce that we can still improve our performance by expanding the number of CPU workers. Since the GPUs are often waiting on the CPUs for work. However, we already ran out of RAM and cannot afford more workers.

Table 2: Comparison between baseline inference algorithms of different batch sizes and our MapReduce method via greedy independent set finding.

Batch size	Per iteration (s)
32	346
64	213
128	131
MapReduce	**20**

6.3 Additional Analysis

Furthermore, the competitiveness of our accuracy suggests that our optimization does not come at the sacrifice of strength. The only possible drawback to our producer-consumer architecture is that it consumes much more memory to store the "produced" intermediate computation graphs for the GPU to consume.

7 Work Distribution

All the work has been assigned 50-50. See the following table for details.

Table 3: Work Distribution

Timeline		**Work**	**Assignment**
Siyang	Yue		
Week 1		Distributed CPU Workload	Siyang
	Week 1	Distributed GPU Workload	Yue
Week 1.5		Training& Debug	Siyang & Yue
Week 2		Report PinSage Training Results	Siyang & Yue
Week 2.5		Map& Reduce Inference	Siyang & Yue
Week 3		Analyze Results for Presentation	Siyang & Yue

References

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983. ACM, 2018.