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.

We plan to implement a web-scale recommender system based on graph convolutional network. Due to the size of the problem, the graph be stored across multiple NVIDIA GPUs and possibly even computers. Our goal is to achieve speedup over the baseline (GraphSAGE Hamilton et al. [2017]) model in distributed training. We will firstly work toward adapting the baseline for distributed training. Then optimize its performance by reducing communication and improving cache locality.

1 Introduction

We attach some introduction from Ying et al. [2018] for the sake of completeness:

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)

1.1 Challenges

The main challenge 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

*Project homepage: qusiyang.github.io

Table 1: Specs on the taobao dataset

Dataset	# user vertices	# item vertices	# user-item edges	# item-item edges
Taobao-small	147,970,118	9,017,903	442,068,516	224,129,155
Taobao-large	483,214,916	9,683,310	6,587,662,098	231,085,487

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 furhter identify the following problems:

1. The problem is still treated as a central problem. 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. GPU workers have to push/pull full gradient vectors to/from the central storage after each batch, which wastes a lot of time.

1.2 Sketch

Our first step is to actualize what's proposed in PinSAGE (Ying et al. [2018]). Through this step we will also be able to further identify more inefficiencies of the proposed method in real-world applications. As we mentioned, most of the implementation details have been omitted in the paper, so we come up with our own guidelines:

- Keep the implementation as claimed in the original paper.
- Use message passing protocol to coordinate computation across different workers.
- Treat the CPUs and GPUs of each worker as if they are in separate machines.
- Perform map-reduce at inference time to gain speedup.

If we were able to achieve good results on the previous step, we will proceed with the following proposed improvements:

1. Fully exploit the locality of the distributed systems by maintaining a local lock-free queue. Allowing the CPU workers to push computation graphs to local que.
2. Create local tree-like weight sharing to allow delayed gradient synchronization. Reducing access to the central CNN.
3. Implement gradient compression by Han and Dally [2018] to reduce commuication.

2 Experiments

2.1 Dataset

We will be working with two of the largest datasets available online. The taobao dataset (Table 1) in Yang [2019], and another public graph dataset. We will also include the datasets available in Hamilton et al. [2017] for comparison.

Table 2: A detailed sketch of our plan

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
	Week 2.5	Gradient Compression	Yue
Week 3		Analyze Results for Presentation	Siyang & Yue

2.2 Resource

We would like to use Bridges' GPU resources provided by Pittsburgh Supercomputing Center. We will start from the code base on GraphSAGE and AliGraph, and use the papers on Deep Gradient Compression and PinSage. We will use User Behavior Data from Taobao for Recommendation as our dataset.

2.3 Platform

The scale of this project necessitates a large number of threads be used for parallel and distributed training. Therefore, we will use CUDA for localized forward propagation, some form of message passing to gather accumulated gradients across GPUs and worker machines, and CUDA for back-propagation.

2.4 Schedule

Week of 10/30: Finish proposal
Week of 11/6: Read and understand the code for GraphSAGE and AliGraph
Week of 11/13: Finish a preliminary/working implementation of parallel graph convolution network
Week of 11/19: Try caching neighbors of important vertices. Also implement LRU caching and random caching of vertices. Measure performance optimizations.
Week of 11/26: Implement gradient compression. Measure performance optimizations
Week of 12/3: Implement lock-free sampling and measure performance optimizations
Week of 12/10: Begin and finish report and prepare for presentation

2.5 Project Checkpoint (due Monday, November 18th)

• One to two paragraphs, summarize the work that you have completed so far.
We have decided to use the Amazon product dataset for our project on parallel optimizations of web-scale recommender system. This dataset contains product reviews and metadata from Amazon, including 142.8 million reviews spanning May 1996 - July 2014. This dataset includes reviews (ratings, text, helpfulness votes), product metadata (descriptions, category information, price, brand, and image features), and links (also viewed/also bought graphs), which allows us to construct a graph and node embeddings for training. We have pretrained two embeddings per node. One uses product image trained using Densenet, and the other uses product descriptions trained using BERT (Devlin et al. [2018]).

We have also come up with the general architecture of our system and started on the implementation. We first use METIS to shard the graph and feed each partition to a CPU to sample a neighborhood. These sampled neighborhoods will each be passed into a GPU for training. We will then sum the gradients from all GPUs to adjust the weights of the network.

2.6 Goals & Deliverables

See (Table 2) for a fine-grained timeline.

1. Implement a parallel and distributed version of the GraphSAGE training algorithm via METIS partitioning. (METIS, Done)
2. Establish a method to calculate importance scores for each vertices, ~~and cache neighbors of important vertices to reduce communication. Also implement other caching techniques such as LRU/random caching for comparison.~~ (We found that the training time (GPU) dominates, better cache would very likely result in negligible improvement)
3. Implement deep gradient compression to further reduce communication.
4. Sampling is required in a large scale network for back-propagation.
5. Measure the performance improvement both in terms of speedup and recommendation quality using F-1 scores.
6. Adopt the above optimizations to the PinSage model. We will show some speedup graphs at the poster session.

• What do you plan to show at the poster session? Will it be a demo? Will it be a graph?

- Speedup Graph
- Table of Recommendation Accuracy wrt to baselines.
- Live Demo of the Recommendation System.

• Do you have preliminary results at this time? If so, it would be great to included them in your checkpoint write-up.
We do not have any preliminary results yet.

• List the issues that concern you the most. Are there any remaining unknowns (things you simply don't know how to solve, or resource you don't know how to get) or is it just a matter of coding and doing the work?
Implementing work distribution on CPU and GPU is just a matter of coding. The only uncertainty comes from implementing gradient compression as there is no publicly available code. We will try to figure out the implementation using Song Han's paper.

References

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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

Song Han and William J. Dally. Bandwidth-efficient deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 147:1–147:6, New York, NY, USA, 2018. ACM.

Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3165–3166. ACM, 2019.

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.