Distributed Training of DLRM

(Group 10)

Ushmal Ramesh
uramesh2@wisc.edu

Varsha Pendyala
varsha.pendyala@wisc.edu

Varun Thumbe
thumbe@wisc.edu

1 Introduction

Recommendation models are used in workloads where there are a set of 'Users' that in some way 'rate' a set of 'Products' and the problem is to predict the rating behaviors. Examples of such workloads include a wide range of scenarios from suggesting a list of recommended movies to a user on NetFlix or a list of recommended books on GoodReads to predicting whether an ad will be clicked or not. Approaches to tackle recommendation workloads have evolved over time. Initially the approach was to cluster the users based on certain criteria and for each new user, recommendations were picked from the statistics of the cluster the user belonged to. Later approaches involved Machine Learning models that were run on features extracted from the user and product datasets. In June 2019 Facebook released DLRM model which combines the above two approaches [10]. The model is used to handle recommendation workloads like Ad CTR predictions. The current DLRM implementation[10] and it's bench-marking[6] is focused on a single node multi-GPU setup and the existing opensource benchmark of DLRM does not handle the case where model needs to be trained on large datasets distributed over multiple servers. Our project is focused on designing and development of a distributed version of DLRM model.

2 Background

The compute patterns in recommendation models as compared to CNNs and RNNs are very different owing to the presence of both categorical and dense features. The DLRM model maps the categorical features into embedding vectors using memory intensive embedding tables learnt with the training data. Width of these embedding vectors is chosen by the designer and the cardinality of each categorical feature determines the size of their corresponding embedding table. So, the inputs are mapped from a high dimensional feature space into a lower dimensional one using learned embedding tables.
The idea of embedding feature categories this way has two-fold advantages - Dimensionality Reduction and Capture of inter-relationships between categories. This is similar to word embeddings where the embeddings of the words 'cat' and 'dog' are mapped closer than unrelated words like 'sky'. Here each feature has a set of categories which can be considered as its 'vocabulary' and the individual categories are embedded in a reduced dimensional space. Even though embedding reduces the dimensions, the tables are still considerable in size and the number of categories is high. Since the cardinality of some of the features can be very high, especially for Ad CTR datasets, the embedding tables tend to be of very high size, usually multi-GBs. The numerical features are processed through a 'bottom' MLP to map them into dense feature vectors of the same width as the embedding vectors. As seen in the Figure 1, the dense feature vectors from both categorical and numerical inputs are then passed through a feature interaction layer and the resulting interactions are then passed to a top MLP for generating the final binary output. We worked with the same interaction operators provided in the original DLRM code - dot-product and concatenation operators.

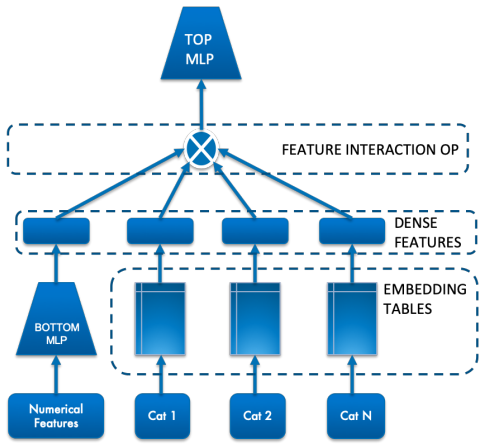

Figure 1: DLRM model

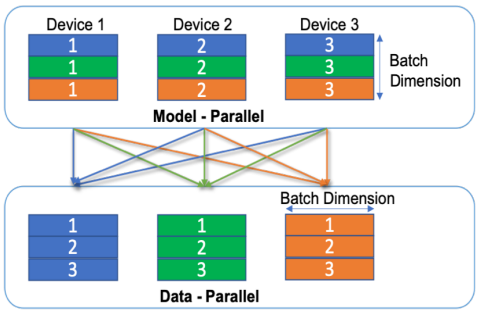

Figure 2: Butterfly shuffle

3 Design

DLRM uses model parallelism to avoid replicating whole set of embedding tables on every GPU device and data parallelism to enable concurrent processing of the samples in FC layers. It is done through a mechanism called butterfly shuffle of the embedding vectors as shown in Figure 2. The figure shows three categorical features sliced across three GPUs. When the input sample arrives, the relevant embedding table lookups are carried out and the embedding vectors corresponding to the various feature categories are fetched from different GPU devices as needed. The MLP parameters are replicated across the GPU devices and therefore need not be shuffled.

While this butterfly shuffle makes sense for a single node, transferring such embedding tables across the nodes in a cluster becomes expensive and could be a bottleneck. Since it is the interactions between the pairs of learned embedding vectors that matter and not the absolute values of the embeddings themselves, we hypothesized that we can learn the embeddings in different nodes independently to result in a good model. This allowed us to save on network bandwidth by synchronizing only the MLP parameters and learning the embedding tables independently on each of the server nodes.

In order to speed up training, we have implemented sharding of the input dataset across the cluster nodes such that both nodes can process different shards of the data concurrently and therefore do more progress in lesser time than a single node.

3.1 Dataset Selection

Since the Criteo Ad CTR dataset was too large to carry out multiple experiments across parameters like batch_size, number of GPUs on a node etc, we decided to look for public datasets having similar properties, i.e., datasets with good mix of categorical and numerical features in the domain of recommendation workloads. We decided upon using the Yelp Reviews dataset which is publicly available for download. (Link). The dataset is split into multiple parts containing User and Business related data. The data consists of a large number of businesses with their ratings by different people and other attributes and user data on reviews, friends count etc. We decided to formulate our recommendation problem as follows: Predict whether a given business would be rated above 4 stars or not. This is a Binary Classification problem similar to predicting whether an ad will be clicked or not.

We processed the Yelp dataset by joining the Business attributes table with the Users attributes and then using a groupby on the business ids, we aggregated the user attributes per business. In the resulting dataset, we identified and retained a set of categorical and numerical attributes to be given as input to our model.

3.2 Experiment Setup

The design was implemented by modifying the Pytorch version of the Facebook's DLRM code from [2] to add code for initialize, manage and maintain a cluster. The number of nodes to spin up are specified and one of the nodes is designated as the master node. The master node collects the gradients of MLP parameters from slave node and itself. We use the torch.distributed.all_reduce API to synchronize these gradients. We created a script that initiated an ssh connection and executed the python job for the distributed DLRM training in each of the nodes.

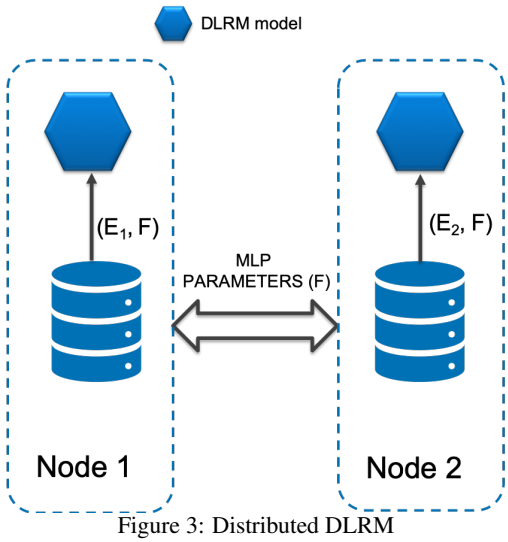

Figure 3: Distributed DLRM

The experiments were initially carried out on GCP and then moved to cloudlab cluster of two c4130 nodes. c4130 nodes have at least 2 GPUs each (The c4130 machines in Wisconsin zone have 4 NVidia V100 GPUS whereas in the Clemson zone they have 2 K40 GPUs).

The dataset is replicated on both machines in our current setup, however we create partitions of the data such

that each node will pick only its share of the data. This is done by assigning different partition to each node based on it's rank in the cluster [1].

We ensured that MLP parameters were synchronized after the all_reduce step in each iteration by monitoring their values for some of the experiments. Also we verified that the embedding tables learnt were different in both nodes as these are not synchronized and the nodes work on different shards of the input dataset. We conducted a simple experiment to see the effect of data partitioning on the embeddings learnt on two nodes in a cluster. Results in Figure 4 were obtained by running the

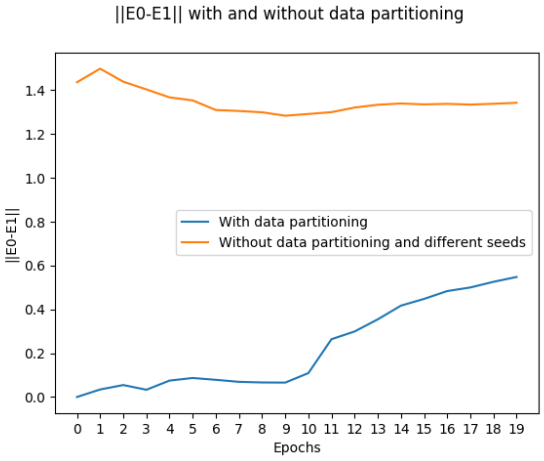

Figure 4: Norm of difference in embeddings on two nodes in a cluster

code on yelp dataset with 512 batch size for 20 epochs. Accuracy and other measurements for these experiments are detailed in Section 4.

When the data is not partitioned and a different random seed is set on each of the nodes, as expected, the accuracy of the model on both nodes converged to a similar value. However the norm of the difference between embedding vectors on both nodes did not converge to zero. Instead, it settled at a non-zero value. This implies, entirely different sets of embeddings can still lead to similarly performing models. We also observed how the norm of difference between embeddings on two nodes changes when data is partitioned - it starts from zero and increases and the plateaus to a constant value.

We therefore infer that irrespective of whether the data is partitioned or not, the norm of the difference in embeddings across nodes settles to a non-zero value. So, it is possible that even though the embeddings are not exactly the same on both nodes, the resulting features after interacting with dense features have some invariance properties due to the synchronized MLP parameters. Since the DLRM uses the interactions of the embeddings rather than the embeddings themselves, we were able to achieve good models even through embeddings are not synchronized across the nodes. Without this, the distributed implementation would be too network bandwidth expensive.

4 Evaluation

In order to evaluate our implementation, we ran few experiments by varying parameters such as batch size, number of GPUs per node and cluster mode (Single node vs Multi Node). We created a bash script that launches jobs by changing these parameters. Note that the Single mode here refers to a single node running our implementation of DLRM. The measurements required from these jobs were output to a file and parsed later with a python script for generating the charts shown below. In the subsections that follow we discuss those charts and the inferences made thereof.

4.1 Accuracy Measurements

The line charts in Figure 5 and Figure 6 show the training accuracies obtained while running the Distributed DLRM in Single mode vs Cluster mode with 2 nodes and 2 GPUs per node.

=

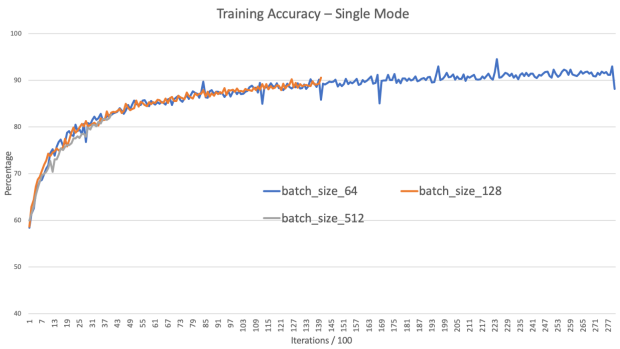

Figure 5: Training accuracy - Single Mode

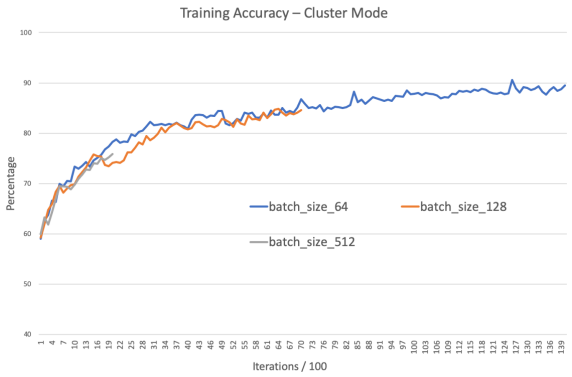

Figure 6: Training accuracy - Cluster mode

From these measurements we infer that the Cluster mode implementation takes around half the number of iterations to reach comparable levels of accuracy as the Single mode implementation. We attribute this speedup to the sharding of the input dataset that allows the nodes to make progress concurrently on different input shards.

4.2 Time Measurements

We now compare the time taken per iteration for Single mode vs Cluster mode. The results are given in Figure 7 and Figure 8.

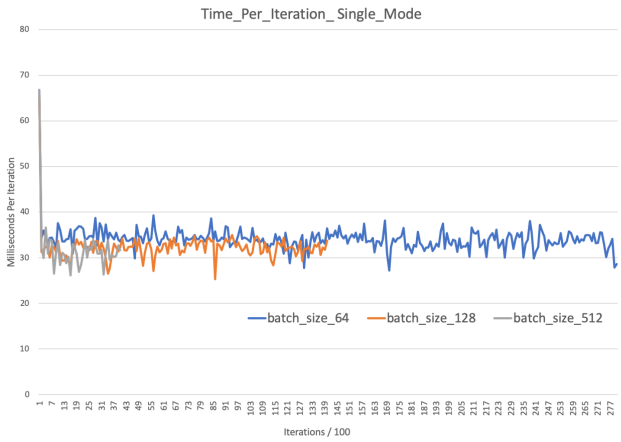

Figure 7: Time per iteration - Single mode

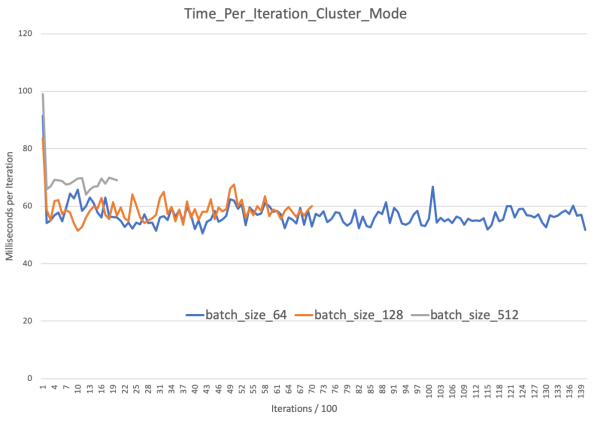

Figure 8: Time per iteration - Cluster mode

From the time measurements we observed that that cluster mode took slightly higher time per iteration than the Single mode. We believe that this extra time is due to the all_Reduce operation. In order to understand more, we added some more time measurements in our code to get more granularity in our measurement data. The figures 9 and 10 provide a break up of the computation time per iteration into sub operations. The sub operations are as follows:

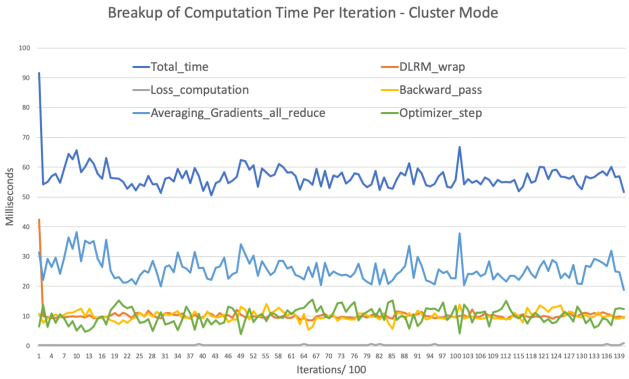

Figure 9: Breakup of computation time per iteration - Single mode

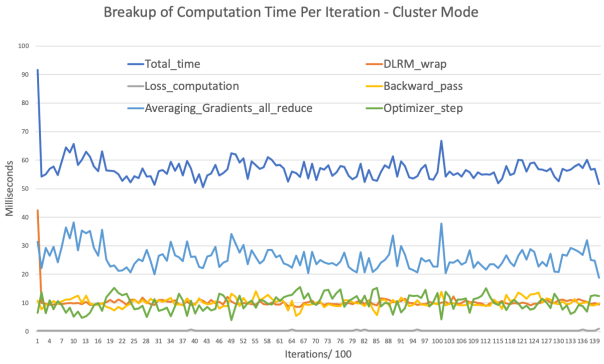

Figure 10: Breakup of computation time per iteration - Cluster mode

- **DLRM_Wrap** - This involves copying tensors in the model from the host (CPU) to the GPU devices and then doing a forward pass.
- **Loss_Computation** - Computation of loss during training.
- **Backward Pass** - Backprop operation during training.
- **Average_Gradients_all_reduce** - This is the all_reduce operation that averages the gradients received from all nodes and then scatters them to all nodes.
- **Optimizer_step** - This performs a parameter update based on the current gradient (stored in .grad attribute of a parameter).

Thus for cluster mode, the all_reduce op takes a major chunk of the time spent in any iteration. This is not the case for Single mode and therefore Cluster mode takes longer per iteration. Figures 11 and 12 summarize the average time spent in each operation for single and cluster modes. This proves our assertion that all_reduce

costs cluster mode additional time per iteration. However, the overall average time per iteration of 35ms for single mode vs 55ms for cluster still shows that cluster mode is much beneficial to be adopted for training DLRM as it takes exactly half the number of iterations for just a minor increase in the average time per iteration. We think that the advantage of distributed training will outweigh the extra time cost of all_reduce when the number nodes increase further than 2. Due to limited resources, we could not experiment with more than 2 nodes in this project.

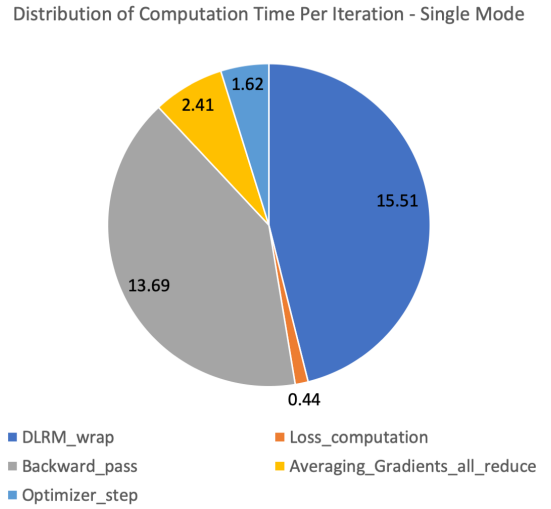

Figure 11: Distribution of computation time per iteration - Single mode

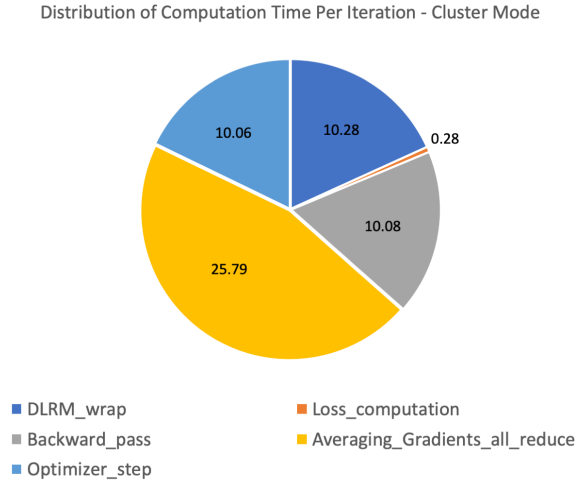

Figure 12: Distribution of computation time per iteration - Cluster mode

4.3 Embedding size

The original embedding size and the MLP parameter dimensions were set according to the Criteo dataset. They were considerably high since Criteo dataset had very high-cardinality features. We found that reducing the embedding size from 16 to 4 and bottom MLP parameter dimensions from 512-256-64-16 to 64-32-16-4 as well didn't have significant impact on testing/validation accuracy as shown in Figure 13 and 14 for a batch size of 512.

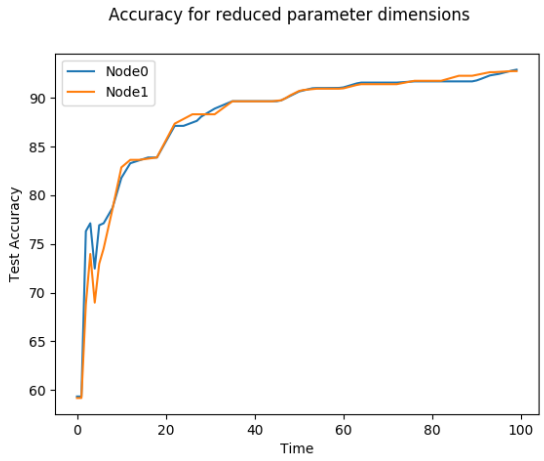

Figure 13: Test accuracy with embedding size 4 and MLP parameters reduced by 1/512th factor

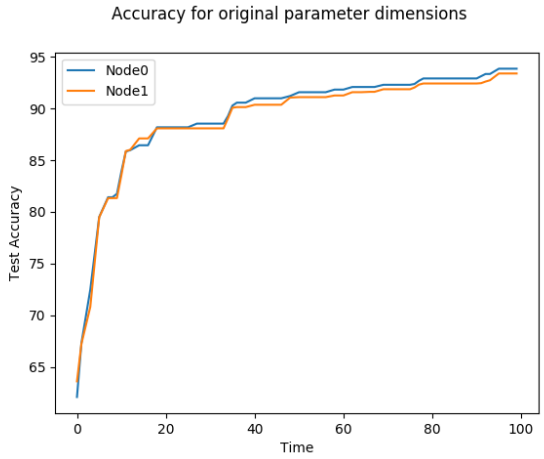

Figure 14: Test accuracy with embedding size 16 and original MLP parameter dimensions

We also see in the figures that, although both the nodes operate on different parts of the data, they have similar trend in the Testing accuracy irrespective of parameter dimensions. For instance, we can see in Figure 13 that both the nodes observe similar dips and spikes in accuracy at around epoch number 5. This similar trend can be due to syncing of the MLP parameter nodes. This could indicate that, even after data partitioning, both the nodes see similar examples which in turn implies interactions learned are similar in both nodes.

4.4 Single GPU experiments

We also ran a number of experiments with Single GPU machines. The results of comparisons between single mode and cluster mode mostly matched the inferences already drawn from multi-GPU experiments. We observed that the Single Mode with 1 GPU took lesser time per iteration (around 8ms) than Single Mode with 2 GPUs. This is due to the fact that there is no Butterfly Shuffling of embedding tables across devices and also the fact that the Yelp dataset we chose is of a lesser size than Criteo dataset. Similarly, Cluster mode with 1 GPU per node also takes lesser time per iteration (around 11 ms) than Cluster mode with 2 GPUs per node for the same reason. Also, as before, Cluster mode with 1 GPU per node takes lesser number of iterations and therefore makes progress faster than a Single Node with 1 GPU.

4.5 Issues faced

During our experiments we faced a number of technical Issues. We summarize them below.

Bug in DLRM Code: During our initial experiments with our model, we observed that our model worked correctly with upto 2 GPUs in each node and when we tried to perform experiments with a higher number of GPUs, the experiment crashed with a segmentation fault. Detailed investigations yielded no results and finally we reverted to the original DLRM code to verify if that model worked with more than 2 GPUS. When we ran the original DLRM code on 4 GPUs, the job crashed again with segmentation fault. We tried investigating further about the root cause of the bug using tools like Xfaulthandler but could not resolve it and raised a bug report at the DLRM project github page. (Link).

Pytorch Version : Initially we were running our experiments on 2 c4130 nodes in the Wisconsin cluster. The Wisconsin cluster has 4 tesla V100 GPUs in each c4130 node. However due to reservation restrictions, we had to move our experiment to the clemson cluster of 2 c4130 nodes. The clemson cluster has 2 K40 GPUs in each of its c4130 nodes. After switching to the new cluster, our code broke with the error 'no kernel image is available for execution on the device'. Upon investigating we learned that Pytorch had dropped support for Tesla K40 and hence we had to downgrade our PyTorch version to 1.3.0 to resolve this issue.

5 Related Work

Synchronization and Fault tolerance: To understand the scale of distributed training of DLRMs, it is useful to think of parameter server architecture[9] that enables training of very large models. The key features of parameter server namely asynchronous model update, user-defined filters, KKT filters could be of use if we eventually decide to replicate and synchronize both learned embeddings as well as FC parameters across multiple servers. However, since FC layer parameters unlike embedding vectors are not memory intensive, they can be updated in synchronous manner without very high network overhead by using primitives similar to those in tensorflow[3].

As the current DLRM implementation is available in PyTorch, we can perform periodic checkpointing over multiple servers to achieve scalable and fault-tolerant implementation.

Data and Model Parallelism: In [7], the authors discuss ways to parallelize/distribute deep learning in multi-core/distributed setting.

A more comprehensive search space of parallelization strategies for DNNs called SOAP is introduced in [8]. It includes strategies to parallelize a DNN in Sample, Operation, Attribute and Parameter dimensions. The authors propose FlexFlow, a deep learning framework that uses guided randomized search of the SOAP space to find a fast parallelization strategy for a given model graph and device topology.

Vector embeddings: Language modeling research has recently attracted a lot of effort in solving word embedding alignment[5], bilingual mappings of word embeddings[4] to build single better embedding from existing ensembles of structures. All these efforts essentially try to unify the embeddings independently learnt on different corpus of data. In a broad sense, these language modeling challenges are analogous to the challenge of reconciliation of independently learned embeddings in DLRM.

6 Future work

Through our experiments on Yelp dataset we have observed that learning embedding tables on different shards of data independently across the nodes still achieves comparable performance as a single node case. So, we believe it would be an important exercise to see if the same behavior can be seen with other datasets of varied size and complexity.

Although we were able to achieve good model accuracy with our design decisions, it came at the cost of high GPU compute on one of the GPU devices on each node. We attribute this behavior to the all_reduce operation happening on GPU:0 of each of the nodes in our cluster. It would be an interesting future work to come up with ways to reduce this load imbalance across the GPUs on a node.

As we discussed, the current DLRM code base doesn't support 4 or more GPUs while training. Thus it is important that this bug is corrected so that recommendation models in the future can be trained on scaled-up architectures as well.

In order to gauge the effect of synchronizing embedding table parameters on network bandwidth utilization, we attempted to transmit them over the network during one of our experiments. However, we found that NCCL backend that we used for the distributed mode only supports distributing dense features across the cluster and therefore synchronizing the sparse embedding tables were not supported. Upon further investigating, we learned that switching to GLOO backend would add support for synchronization of sparse data. Hence we believe it would be a good exercise to switch to GLOO backend and measure the impact of synchronizing embedding tables on training in terms of network usage. Also, it might be a good idea to compress these sparse features before distributing across the cluster and measure the training performance. We could not perform these experiments due to time restrictions.

Owing to limited computational resources we have evaluated our design with a cluster of only two nodes. However it would be a good idea to understand the differences in network usage patterns between synchronous and asynchronous all_reduce operation as the number of nodes in the cluster increase as it may provide some insights on choosing the best backend for the distributed training.

We extended the idea of DLRM from single node multi-GPU set up to multi-node multi-GPU set up. However, for our implementation to scale up, there is a need to add a framework layer to help achieve fault tolerance in case one of the nodes fails in the cluster mode. Also, a resource manager layer like Mesos or yarn can be added to automate resource selection process in an industry environment.

References

[1] torch.distributed. https://pytorch.org/docs/stable/distributed.html#initialization.

[2] Dlrm. https://github.com/facebookresearch/dlrm, 2013.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] M. Artetxe, G. Labaka, and E. Agirre. Learning principled bilingual mappings of word embeddings while preserving monolingual invariance. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2289–2294, 2016.

[5] S. Dev, S. Hassan, and J. M. Phillips. Absolute orientation for word embedding alignment. *CoRR*, abs/1806.01330, 2018.

[6] U. Gupta, X. Wang, M. Naumov, C.-J. Wu, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, B. Jia, H.-H. S. Lee, et al. The architectural implications of facebook's dnn-based personalized recommendation. *arXiv preprint arXiv:1906.03109*, 2019.

[7] V. Hegde and S. Usmani. Parallel and distributed deep learning. In *Tech. report, Stanford University*. 2016.

[8] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[9] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[10] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.