Ember: Asynchronous Dynamic Data Serving for PyTorch Distributed Training

Patrick O. C. Araújo[1], **Fábio T. Ramos**[1], **Jhonata M. da Costa**[1], **Mario Drumond**[2], **José Augusto M. Nacif**[1]

[1] Universidade Federal de Viçosa (UFV) – Florestal – MG – Brasil
[2] Huawei – Switzerland

{patrick.araujo, fabio.ramos, jhonata.miranda, jnacif}@ufv.br

mario.paulo.oliveira@huawei.com

Abstract. *Advancements in natural language processing (NLP) and computer vision (CV) have led to substantial growth in data and model sizes, often requiring the distribution of the model and data across multiple GPUs and machines to accelerate training. Existing tools such as PyTorch Distributed Data-Parallel (DDP) operate at a low level of abstraction, requiring vast knowledge of distributed training. Therefore, users must adapt workflows to rigid tools or rebuild the entire code for each model and implementation, which can lead to performance issues. We introduce Ember, a customizable distributed training framework with new data distribution mechanisms. Ember utilizes remote procedure calls to decouple data loading logic from model synchronization, ensuring efficient computational resource utilization while maintaining accessibility and customization. To evaluate Ember's performance, we compare its core training features with Ray, a powerful distributed framework, to ensure that our functions can be competitive with state-of-the-art implementations. The results of testing Ember use on datasets and models show that the framework achieves its goals, reducing GPU idle time and optimizing data transfer, achieving competitive training times with our baseline while utilizing its new asynchronous services with up to 36% less memory usage.*

1. Introduction

Advancements in state-of-the-art machine learning, such as transformer [Vaswani et al. 2023], in the last few years have led to exceptional growth in training datasets and model sizes. For instance, natural language processing (NLP) and computer vision (CV) models have reached the multi-trillion mark in the number of parameters [Pritam Damania 2023].

The long training time of these models steers solution designers towards using distributed systems for increased parallelization and I/O bandwidth, because training datasets used in such sophisticated applications are easily in the order of terabytes. In some cases, centralized solutions are not feasible, because data are inherently distributed or too large to be stored on a single node [J Verbraeken 2020].

PyTorch [Shen Li 2020] and TensorFlow [Abadi et al. 2015], two widely used machine learning libraries, provide basic mechanisms for users to build custom distributed

systems for neural network training. However, training large-scale neural networks demands expertise in distributed systems in addition to familiarity with the libraries and neural network design, making the process complex, time-consuming, and reliant on specialists across multiple domains.

This scenario creates development and performance bottlenecks, leading to a waste of computational resources and negatively impacting the resulting models. Among the issues caused by such complexity, there are possible mismatches and desynchronization between nodes, as well as crashes and inefficient resource utilization [J Verbraeken 2020, Y Gao 2024, A Audibert 2023].

To minimize these issues, Horovod [Alexander Sergeev 2018] and Ray [Moritz et al. 2018] provide their own distributed training systems built on top of these libraries. However, when using PyTorch Distributed Data Parallel (DDP), these frameworks distribute the model and datasets across nodes; each machine keeps the entire dataset in memory, and training can only start once all data are loaded at each parallel worker, therefore requiring larger amounts of memory and disk space and increasing training time.

In this paper, we present Ember[1], a customizable, simple, and efficient distributed training system. First, Ember streams data asynchronously to different nodes using Google Remote Procedure Calls (gRPC) instead of distributing entire datasets before training, as in state-of-the-art approaches. Second, Ember nodes keep only the shards of the dataset that are currently being used by training devices and that will be used soon, significantly reducing memory usage. Third, Ember starts training as soon as the first batch of training data arrives on all GPUs, thus accelerating training by minimizing GPU idling time. Finally, Ember exposes a single-node-like API, hiding all the complexity of dataset distribution.

Experimental results on two different machines, each equipped with one GPU, show that Ember achieves competitive training times, accuracy, and loss compared to the baseline, while reducing RAM usage by up to 36% without significantly compromising final accuracy and loss after training, compared with the state-of-the-art approach, Ray.

We organize the remainder of this work as follows: Section 2 discusses core elements relevant to the discussion of this work. Section 3 compares Ember with the state-of-the-art approaches to distributed training. Section 4 describes Ember architecture and usage. Section 5 depicts the experimental setup, and Section 6 presents the results. Finally, we conclude our remarks in Section 7.

2. Background

In this session we'll discuss important concepts directly linked with the design choices present in Ember, which are driven by a study of the most common bottlenecks faced in distributed pipelines gathered through literature reviews such as [M Aach 2023] and [J Verbraeken 2020].

[1]https://github.com/lesc-ufv/Ember

2.1. Distributed Data Parallel

Figure 1 illustrates the DDP training workflow. Given a neural network model, a dataset, and machine configurations, DDP divides the data and replicates the model across several GPUs, which can be on the same machine with equal GPUs (homogeneous environments) or on different machines (heterogeneous environments). Then, after forward and backward, gradient reduction and weight updating occur across each local model.

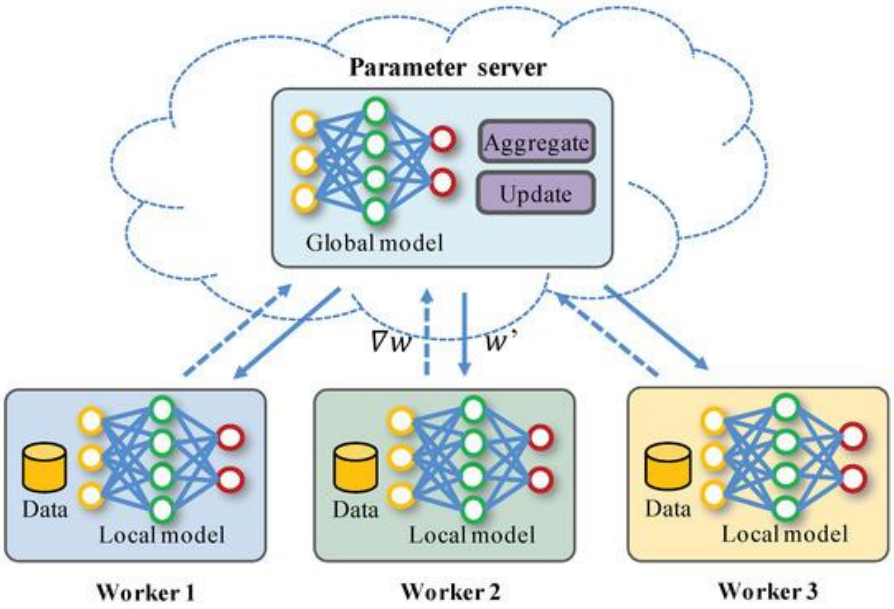

Figure 1. DDP Workflow With Synchronous Data Serving. Image extracted from [Ko and Kim 2021].

2.2. Critical and non-critical datapaths

The process of training in distributed environments has two distinct elements that are crucial in the construction of Ember. In order to synchronize model gradients and intermediate values, each worker node must update its weights and communicate with the other worker nodes synchronously. This part of the pipeline is called a *critical datapath* and contains every step in the workflow that needs constant synchronization and management from the system, such as the DDP `all_reduce` function. Every *critical datapath* point in the environment can lead to bottlenecks since its function must end before any new weight is processed inside the loop.

The other core elements are the *non-critical* parts of the pipeline, such as data loading and checkpoint writing. Ember focuses on creating non-blocking solutions for this section of the training workflow, because its delegation to multiprocessing can lead to substantial gains in training time due to a decrease in barriers that can obstruct the training progress [A Audibert 2023]. The implementation of non-blocking structures isolates the critical datapath to model-specific functions only, helping with the modularization of the training workflow itself, which further isolates the creation of the dataset from the training loop.

Hence, by delegating most of the non-critical datapaths to Ember's built-in functions, users can focus on their own model implementation and training details, with most of the distributed roadblocks out of the way.

3. Related Work

PyTorch and TensorFlow offer ready-to-use APIs for users to implement DDP. Still, they must know how to set up the overall workflow using these APIs, which is costly because of the technical knowledge of neural networks, distributed systems, and API integration, the latter being even more complex due to the sparse documentation. Horovod [Alexander Sergeev 2018] and Ray [Moritz et al. 2018] create a complete DDP workflow, providing a simple tool for distributed training usage.

Each framework comes with its own design choices. Ray offers robust training setups and functionalities, but its system covers the entire training pipeline, including critical and non-critical datapaths. Ember decouples these two important elements to enable users to compose systems as they need, focusing on designing models and optimizing critical datapath-related portions closely related to their necessities.

Horovod, on the other hand, focuses on synchronous data parallelism, working only with the critical path portion of the distributed environment. It doesn't inherently come with its own data serving methods, leaving this portion of the implementation to the user's preference. Ember supplies this demand by creating a system to deal with the non-critical data serving and checkpointing portion of the pipeline.

Additionally, the tf.data service system by Audibert et al. [A Audibert 2023] highlights the benefits of disaggregated data processing from core model training in distributed systems, leading to more efficient data utilization and reduced input bottlenecks. Ember expands on these principles by simplifying distributed training with a standardized data communication protocol across the environment. This focus on usability and flexible model-dataset integration is made so that each node receives only its subset of the original dataset. Thus, we focus on reducing bandwidth and communication bottlenecks.

4. Ember Design

Ember design focuses on the disaggregation of critical and non-critical datapath elements of the DDP to create a more efficient system by utilizing a client-server approach, where data distribution is delegated to the thread orchestrated by the server process that continuously feeds new batches to each worker node. The workers encapsulate all the critical datapath portions related to the synchronization and update of weights made with DDP built-in tools [Shen Li 2020].

Figure 2 shows the Ember client-server approach, in which a server process orchestrates data serving for each node, ensuring no replication occurs between them, and the next sections present in details each Ember component.

4.1. Server

The first main responsibility and execution on the server side is dataset augmentation, guaranteeing it is done only once and therefore saving time during training. Subsequently, the resulting data goes directly to the serving stage, which initially shuffles and pre-splits the dataset for each worker and then starts serving with an asynchronous pipeline, creating a separate thread according to the workers' requests for populating each queue. At the end of each epoch, the initial step reorganizes the dataset with a new shuffling before serving, so that the models do not memorize data distribution.

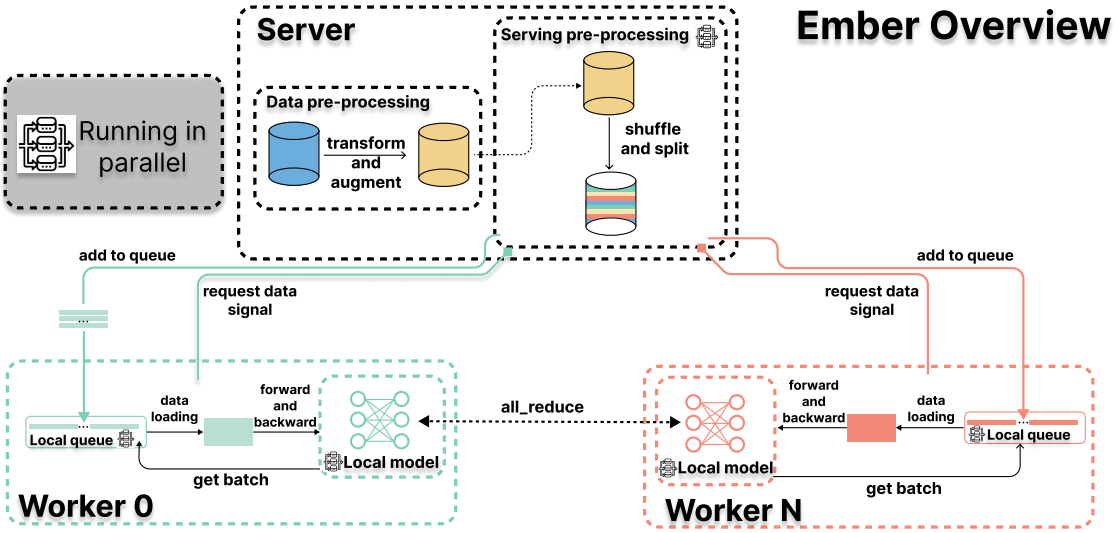

Figure 2. Ember Distributing System Pipeline Overview.

In this way, Ember decouples data transfer from the standard methods applied in DDP and implements its own data distribution with gRPC. Consequently, it maintains consistency in the training process while permitting the early start of each training loop with subsections of the total dataset. Moreover, Ember tries to divide data equally among each worker, guaranteeing the best split between any number of nodes in the training group. This is made based on the pre-configured configuration files passed to the API at the start of training, an important step to maintain consistency and synchronization. Ember handles scenarios where an even split is not possible by cutting the dataset size to the largest divisible amount before shuffling. This way, at each epoch, the portion not used is different, still training on the entire set at the end of the total training loop.

4.2. Worker

With data ready for training, each worker (client) requests k samples from the server according to its partition, creating a thread for the upcoming continuous reception of chunks of data. As soon as the first data chunk arrives, a queue starts to be filled with the incoming data. Once the queue contains a batch of size b, the first b samples are collated and sent for training the model.

In this case, a synchronization mechanism prevents workers from starting the next iteration before all other nodes in the current group have already processed an equal number of batches. This behavior ensures that there are no mismatches or desynchronizations in the weight updates that could compromise the model's final results.

Moreover, it is worth noting the benefits of asynchronous data serving, as the queue continues to be populated with data from the server while training occurs, thus avoiding bottlenecks in the process.

4.3. API and usage

Our framework features a set of declarative API exposed functions that can be used to set up a training loop with minimal configuration. At the same time, it maintains all core steps of the entire training pipeline open to fine customization by the user.

The API for running a minimal setup requires two configuration files: one that specifies to the server the total number of working nodes, called `world_rank`, and dataset-related information, such as each transformation that needs to be applied, and a string specifying the path of the hard drive location of the files. The second configuration file is passed to each node and contains its index on the cluster, paired with the batch size for the training loop.

Ember usage aims at minimizing the learning curve of using distributed training with PyTorch by making its usage as close as possible to a normal training pipeline without distribution. Algorithm 1 illustrates a normal training setup requires specifying the model, defining helper functions such as optimizer, criterion, and learning rate, and then loading the dataset into a data loader before iterating through batches.

Algorithm 1: Example training pipeline without distribution.

```
torch.cuda.set_device(gpu);
model = ExampleModel();
criterion = ...;
optimizer = ...;
learning rate = ...;
dataset = load_dataset();
dataloader = dataloader(dataset, ...);
for i in range epochs:;
...;
```

The same pipeline presented in Algorithm 1 can be used with Ember by importing the framework and calling its functions in three steps: defining training parameters, spawning the training group, and initiating the data loading thread, respectively.

Algorithm 2: Example training pipeline with Ember

```
ember.start_group(parameters);
torch.cuda.set_device(gpu);
model = ExampleModel();
model = DDP(model, parameters);
criterion = ...;
optimizer = ...;
learning rate = ...;
dataset = ember.load_dataset();
dataloader = dataloader(dataset, ...);
for i in range epochs: ;
...;
```

The setup presented in Algorithm 2 abstracts much of the work necessary to handle the asynchronous batch serving through its internal API, requiring the user to only call the methods responsible for handling the data distribution in the background, maintaining the training code more closely related to a simpler training pipeline.

4.4. Data Partitioning

Ember removes the need to replicate data by utilizing a server-client approach with remote calls from gRPC middleware, in which the server process running on a single machine containing the data set acts as a storage point for the worker nodes. The worker nodes can send a request with their rank and receive a part of the data set based on the total number of workers in the cluster. This approach ensures no replication of unused data, addressing a key problem that can slow training in distributed environments [Y Gao 2024, M Aach 2023].

The server process is responsible for data preparation, where pre-processing is performed by defining transformations and data augmentation. Each node sends a request to the server, awaiting a part of the data set. The server then divides the total number of samples by the total number of nodes received in the request through the `world_size` variable, sending back to the node only its part of the training data and removing the necessity to replicate the entire set on each node.

A synchronization mechanism makes each node wait to receive information from every other node in the environment before starting each training loop. This step is necessary due to the synchronous nature of DDP, making the entire training process consistent across nodes to ensure proper utilization of gradients between workers [Shen Li 2020, M Aach 2023].

4.5. Dataset Streaming

A heterogeneous environment can have differences in hardware between nodes, including available memory and cache memory. When dealing with huge data sets, it is necessary to implement functions that can avoid large data loadings into RAM. For that purpose, Ember relies on its own dataset streaming logic, focused on maintaining small portions of the dataset in memory at a time, avoiding leaks and providing consistency even with machines with different configurations. This setup does not require copies of the data on a local hard drive, since the queue for the next epoch will refresh the entire set.

The dataset streaming queues chunks of data received from the server, as Figure 3 shows, which are then loaded as batches using the pre-configured parameters of the training loop, training the model with the current batch and then freeing memory for the next ones. This process requires synchronization between nodes and server, but can grant low memory usage at the cost of increased bandwidth use.

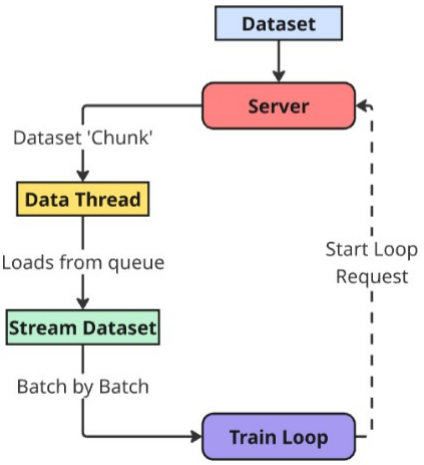

Figure 3. Streaming Dataset Pipeline.

4.6. Server workload

A new challenge created by the asynchronous chunk serving is the necessity to send data at a constant pace while the workers are training. This process creates a constant load on the CPU of the server machine as shown in figure 4, which needs to process and send data to each node, while maintaining the time consistent with the necessary time to match the workers' needs. With a correct setup of calculations that can be done in the framework, it is possible to define an optimal size of data chunk vs. batch size to maintain the GPU always processing batches without ever emptying the queue.

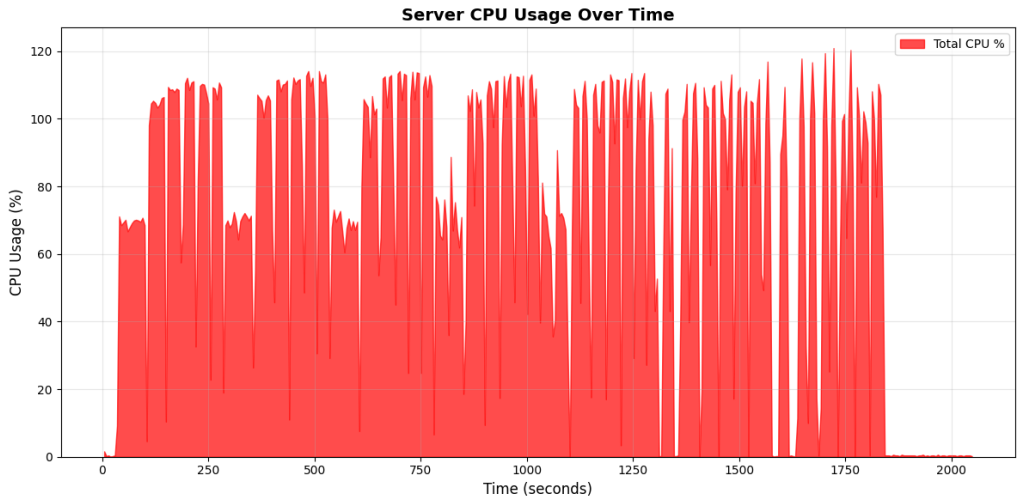

Figure 4. Server CPU Utilization Over Time.

This setup ensures a faster training time, removing one of the major bottlenecks of data loading vs. batch processing [M Aach 2023, A Audibert 2023]. CPU utilization remains constant during the entire training process because the asynchronous streamed dataset approach necessitates a copy of data at each new epoch. It is important to pinpoint that this behavior is specific to the functionalities of utilizing Ember tools for huge datasets that don't fit in memory.

4.7. Model Evaluation Step

During the `allreduce` step at the end of each training iteration, the nodes collaboratively normalize their parameters to ensure consistency in the distributed environment [Shen Li 2020]. Thus, the model's state synchronizes across the environment. As a result, after training, only one node needs to evaluate the model's performance. Node `rank 0` carries out this process, which instantiates a call to the server to retrieve the test set to compute the evaluation of the model and optionally save the checkpoint and any metric associated with the process.

The evaluation requires minimal distributed effort once the processes of synchronizing parameters and model training have finished. Moreover, the framework supports customization in a way that the user uses it in evaluation. Our framework offers a function to request the test data with no parameters needed except for the connection-related; since the process only needs to be done on one machine, no rank and world size parameters are needed, ensuring ease of use. It's possible to adapt the training loop to request validation sets from the server and print each node's parameters and weights since the workers retain local information about their own training steps.

5. Experimental Setup

We have conducted the evaluation of the proposed framework in a distributed environment made of two distinct nodes connected via a LAN network. To ensure a proper scenario in which the tests could be run with consistency and manageability in a heterogeneous system, the tests were performed on distinct machines equipped with an NVidia A40 GPU and one with an NVidia 5090, as Table 1 presents.

Table 1. GPU hardware specifications used in the experimental evaluation.

GPU Model	VRAM	Memory Type	CUDA CC
NVIDIA A40	48 GB	GDDR6	8.6
NVIDIA RTX 5090	32 GB	GDDR7	10.x

We ran each process of the distributed environment using Docker Compose to create isolated containers representing multiple virtual nodes, each operating as an independent instance, as seen in Figure 5. Thus, enabling the evaluation of inter-node communication and workload distribution under simulated conditions allows testing the framework functionalities while ensuring replicability and customizations, since changing the number of nodes and other network configurations was possible through utilizing Docker's built-in functionalities.

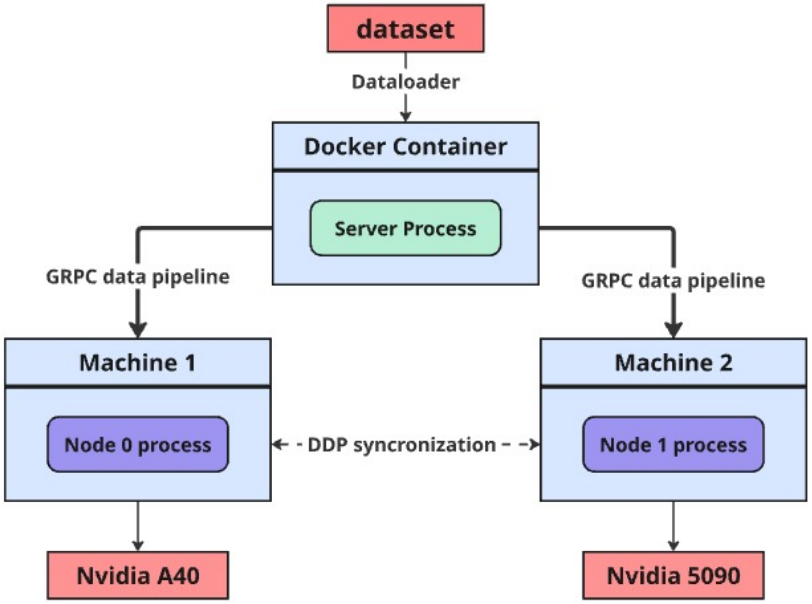

Figure 5. Experimental Distributed Setup with Two Nodes.

6. Results and Discussion

We conducted network tests and analyzed the total network loadout and model on Ember and on our baseline framework, Ray Distributed [Moritz et al. 2018]. We utilized one small model using a small neural network with three convolutional layers designed to train on the MNIST dataset [Deng 2012] to analyze specific parameters and training behaviors and to check for any possible anomaly generated by the system relaxations necessary for asynchronous data serving, which can potentially lead to differences in accuracy or training time [Y Gao 2024, M Aach 2023, A Audibert 2023]. These anomalies, such as noticeable differences in accuracy and loss caused by mismatches in synchronization, differences in training time, or batch count caused by issues in data serving, could be a sign of a problem in the construction of the distributed pipeline.

The main goal is achieve a performance comparable to the baseline Ray distributed pipeline, which is a standard and powerful tool, while adding our layer of streamed data to analyze the impacts and tradeoffs of its usage against a established system. The principle of this comparison is analyzing if a significant loss in training time and accuracy occurs when using a relaxation in data distribution. To guarantee this result, the exact same model parameters, dataset and ambient was used in each scenario, granting that differences only existed in the data transfer layer.

Table 2. Training Performance.

Framework	Accuracy	Loss	Time
Ember	98.86%	0.1334	0:58:05
Ray	98.9%	0.1340	1:06:54

Table 2 contains training performance results, showing that Ember achieves similar training times due to the direct setup of network variables and network connections,

being able to directly connect to the nodes and start data distribution and training earlier than Ray, resulting in slightly faster times. Even with system relaxations and asynchronous data loading, no tangible differences in loss or accuracy can be attested, being perceived only as small variations caused by shuffling and noise applied during training.

6.1. Memory Efficiency

One of the core gains of using the new asynchronous streamed dataset is the capability to use less RAM during the training loop, ensuring the ability to train on datasets larger than the total RAM available on the worker nodes. To analyze the scale of the gain of utilizing this approach, we compared the total RAM utilization during the training loop on the Ember streamed dataset and our baseline without asynchronous serving.

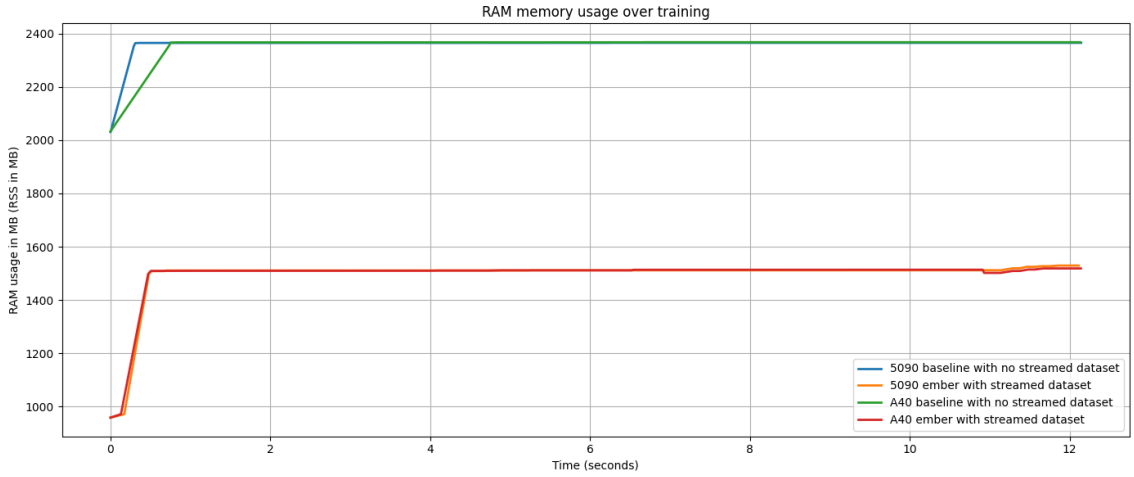

Figure 6. Ember vs. Baseline Memory Efficiency Results.

Figure 6 presents memory usage results, which, by streaming batches from the queue thread and starting the training loop early, we achieve two important goals. First, training starts earlier, as can be seen by the steep rise in memory usage that indicates model loading into memory, which is faster in Ember. Second, we observe the difference in memory usage obtained by not loading the entire dataset simultaneously. Memory usage remains stable since new batches are loaded continuously, ensuring overall usage that is much lower than the baseline.

Table 3. Percentual memory usage reduction.

Metric	Ember (MB)	Baseline (MB)	Reduction (%)
Peak RAM	1519.1	2368.0	35.85
Mean RAM	1512.2	2367.8	36.14

6.2. Model Usage

One of the main goals of Ember is its adaptability to different model architectures and datasets to enable robust utilization across various scenarios. To evaluate its capabilities for use with different models, we trained and compared different architectures on a public-domain dataset. We conducted the tests using the ResNet18 [K He 2015], MobileNetV2

[M Sandler 2019], and EfficientNet [M Tan 2019] models. We initialized the selected models with pre-trained weights from the PyTorch library to leverage transfer learning and accelerate convergence during training.

We conducted the training tests using a dataset comprising 30,000 images of 15 classes of vegetable species distributed among those types [Ahmed et al. 2021]. This dataset comprises images exceeding 224×224 pixels, which provides an opportunity to analyze the communication overhead between the server and workers. The large file size of these images is particularly relevant for evaluating the framework's ability to handle data traffic efficiently during asynchronous batch serving.

Table 4. Performance and efficiency comparison between Ember and Ray.

	Ember			Ray		
Model	Accuracy	Loss	Time	Accuracy	Loss	Time
ResNet18	98.6%	0.1523	**1:52:61**	**98.8%**	**0.0623**	2:10:61
MobileNetV2	97.93%	**0.0975**	**1:55:26**	**98.12%**	0.1175	2:08:37
EfficientNet	**95.91%**	**0.1126**	**2:03:11**	95.67%	0.2126	2:13:09

Ember achieved accuracy and loss results close to those of the powerful Ray framework, showing that asynchronous batch serving can achieve similar accuracy and loss metrics while maintaining very low memory usage, as Figure 6 shows, by not requiring copies of the dataset in memory and relying on good timing between training and data sharing across the asynchronous system. The competitive training times compared with Ray, combined with the 36.14% reduction shown in Table 3, indicate how Ember can be used to maximize GPU work-time efficiency and minimize RAM usage.

To test the impact of correct parameter tuning between batch size and data chunk size, we executed a compound training loop combining different batch and chunk sizes to analyze differences in GPU idle time. The core principle is to minimize this bottleneck as much as feasible, enabling a faster overall training loop. This configuration varies across model and dataset combinations, since different training loops have unique properties. For example, a classification task using images can incur larger data transmission times due to image complexity while using a simpler model that processes batches quickly. In contrast, transformer model fine-tuning on text data can involve faster data transmission but longer batch processing times due to attention mechanisms, depending on hyperparameter configuration.

This test was conducted using the CIFAR10 dataset and a small neural network with three convolutional layers.

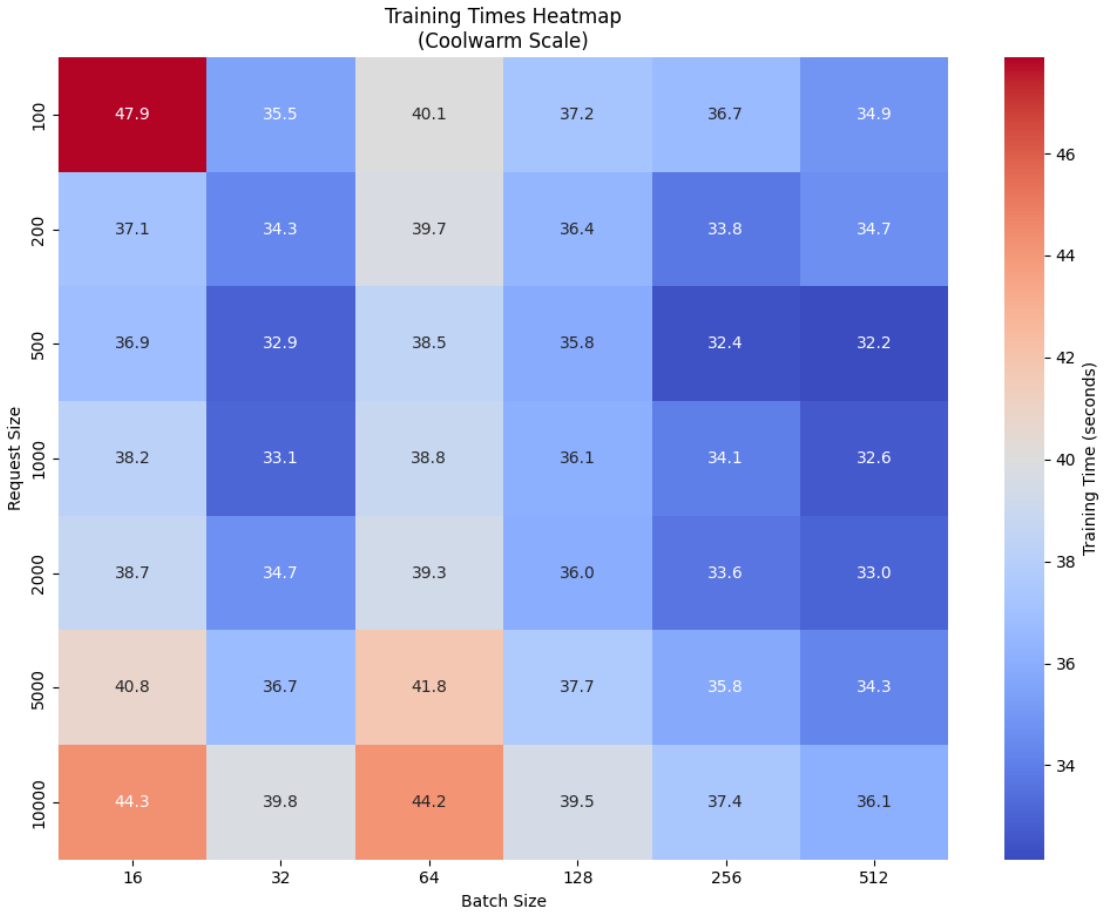

Figure 7. Heatmap Breakpoints Over Different Batch vs. Chunk Sizes.

Analyzing the heatmap presented in Figure 7, we observe a clear pattern indicating training-time behavior based on the relationship between the amount of data transmitted and batch size. With a batch size larger than the chunk size, we obtain the worst times, since each new data transmission from the server introduces a small delay. The best cases for this specific model occur when the chunk size is close to the batch size, as transmission times and batch training times are similar. Increasing the chunk size excessively also creates longer idling times, as workers must wait longer before starting.

From these results, we can extrapolate the behavior to larger datasets and more complex models. A model that requires more time per batch can operate with a larger chunk size, as long as there is always sufficient data available for at least one complete new batch iteration when the previous one finishes. These findings indicate the necessity of a great balance between the chunk size sent by stream and the batch size, which can decrease training time while maintaining the memory usage low. A good starting point as shown in 7 is a batch size at least with batch size equals to half of the chunk size to guarantee a window for the synchronization and training of batches. This pattern can change based on model and connection capabilities, requiring some tuning that can be done analyzing the overall training behavior of a given model.

7. Conclusion

The machine learning field continues to expand, with increasing data sizes and model complexities pushing computational demands beyond what a single machine or GPU can sustain. Consequently, parallelizing workloads across multiple machines is essential for technological advancement and resource optimization. However, implementing distributed environments introduces additional complexities and challenges that make pipeline creation highly scenario-dependent. Moreover, the steep learning curve can present a significant obstacle to implementation and optimization, making training setup for each model or dataset challenging and time-consuming.

This paper introduces Ember, a robust distributed training pipeline that leverages PyTorch DDP and gRPC to create a customizable distributed training framework. The project aims to facilitate the implementation and use of training pipelines that accommodate different datasets and model architectures while achieving competitive results. By decoupling data distribution from model synchronization, we address key challenges in distributed deep learning, such as inefficient GPU utilization and the need for dynamic dataset handling. The framework's modular design simplifies training on diverse datasets and models, ensuring ease of integration and flexibility for developers. Ember significantly reduces GPU idle time and boosts training efficiency through asynchronous data serving, achieving substantial memory usage reduction with its streamed dataset structure while maintaining competitive accuracy and loss metrics comparable to powerful baseline frameworks such as Ray Distributed.

Future research can build upon these core optimization principles by expanding Ember's capabilities with automated tools to identify optimal trade-offs between chunk and batch sizes for each use case. Furthermore, exploring additional system relaxations in other segments of the distributed pipeline, such as node coordination, could enable greater training dynamism and help reduce desynchronization issues.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. Fapemig and CNPq have also financially supported this work.

References

A Audibert, Y. C. (2023). tf.data service: A case for disaggregating ml input data processing.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous distributed systems. Software available from tensorflow.org.

Ahmed, M. I., Mamun, S. M., and Asif, A. U. Z. (2021). Dcnn-based vegetable image classification using transfer learning: A comparative study.

Alexander Sergeev, M. D. B. (2018). Horovod: fast and easy distributed deep learning in tensorflow.

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*.

J Verbraeken, M. W. (2020). A survey on distributed machine learning.

K He, X. Z. (2015). Deep residual learning for image recognition.

Ko, Y. and Kim, S.-W. (2021). Shat: a novel asynchronous training algorithm that provides fast model convergence in distributed deep learning. *Applied Sciences*, 12(1):292.

M Aach, E Inanc, R. S. (2023). Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks.

M Sandler, A. H. (2019). Mobilenetv2: Inverted residuals and linear bottlenecks.

M Tan, Q. L. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. (2018). Ray: A distributed framework for emerging ai applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577. USENIX Association.

Pritam Damania, S. L. (2023). Pytorch rpc: Distributed deep learning built on tensor-optimized remote procedure calls.

Shen Li, Y. Z. (2020). Pytorch distributed: Experiences on accelerating data parallel training.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

Y Gao, Y He, X. L. (2024). An empirical study on low gpu utilization of deep learning jobs.