Distributed Training of Deep Neural Networks: Convergence and Case Study

Jacques M. Bahi[1], Raphaël Couturier[1], Joseph Azar[1], and Kevin Kana Nguimfack[1]

Université de Franche-Comté, CNRS, institut FEMTO-ST, F-25000 Besançon, France
{jacques.bahi,raphael.couturier,joseph.azar,kevin.kana_nguimfack}@univ-fcomte.fr

Abstract. Deep neural network training on a single machine has become increasingly difficult due to a lack of computational power. Fortunately, distributed training of neural networks can be performed with model and data parallelism and sub-network training. This paper introduces a mathematical framework to study the convergence of distributed asynchronous training of deep neural networks with a focus on sub-network training. This article also studies the convergence conditions in synchronous and asynchronous modes. An asynchronous and lock-free training version of the sub-network training is proposed to validate the theoretical study. Experiments were conducted on two well-known public datasets, namely Google Speech and MaFaulDa, using the Jean Zay supercomputer of GENCI. The results indicate that the proposed asynchronous sub-network training approach, with 64 GPUs, achieves faster convergence time and better generalization than the synchronous approach.

Keywords: Distributed Deep Learning · Asynchronous convergence · Subnet Training · Parallelism.

1 Introduction

The work presented in this paper concerns distributed and parallel machine learning with a focus on asynchronous deep learning model training. Model parallelism and data parallelism are the two primary approaches to distributed training of neural networks [1,2]. In data parallelism, each server in a distributed environment receives a complete model replica but only a portion of the data. Locally on each server, a replica of the model is trained on a subset of the entire dataset [3]. Model parallelism involves splitting the model across multiple servers. Each server is responsible for processing a unique neural network portion, such as a layer or subnet. Data parallelism could be deployed on top of model parallelism to parallelize model training further. In a deep convolutional neural network, for instance, the convolution layer requires a big number of calculations, but the required parameter coefficient W is small, whereas the fully connected layer requires a small number of calculations, and the required parameter coefficient W is large. Consequently, data parallelism is appropriate for convolutional layers, while model parallelism is appropriate for fully connected layers.

Nevertheless, data and model parallelism approaches have numerous drawbacks. Data parallel approaches suffer from limited bandwidth since they must send the whole model to every location to synchronize the computation. A huge model with many parameters cannot meet this requirement. Model parallel approaches are not viable in most distribution settings. When data are shared across sites, model parallel computing requires that different model components can only be updated to reflect the data at a specific node. Fine-grained communication is required to keep these components in sync [4].

1.1 Objective

The study has two main objectives. Given a neural network and its partition into sub-networks, the first objective is to establish the mathematical relationship between deep learning on the complete network and deep learning on the partitioned sub-networks. The second is to use this mathematical model to clarify sufficient conditions that ensure the effective convergence, after training of the sub-networks, towards the solution that would have been obtained by the complete neural network. The research question **RQ** is "*whether, by training and combining smaller parts of a full neural network, the convergence of the global solution to the original problem is obtained. In other words how to ensure that weight parameters obtained by training the partitioned parts of the full network reconstitute the correct weight parameters that tune the full network.*"

The mathematical model we introduce is general; indeed, it covers both sub-network training and federated learning approaches, as well as their execution in both synchronous and asynchronous modes. To validate our model, we conduct synchronous and asynchronous implementations of sub-network training with a coordinator and several workers and compares their performance.

Our implementation aims to validate the proposed mathematical model and its necessary conditions, rather than to offer a novel deep learning architecture for distributed deep learning. For this purpose, we present an implementation based on the Independent Subnet Training (IST), an approach proposed by Yuan et al. [5]. The experiments we conducted [1] demonstrate how asynchronous distributed learning leads to a shorter waiting time for model training and a faster model updating cycle.

1.2 Methodology

We introduce a general mathematical model to describe the sequences generated[2] by sub-networks partitioning-based approaches. The model considers the depen-

[1] Experiments are conducted on GENCI's Jean Zay supercomputer in France, which contains 64 GPUs

[2] The generated sequences, or produced sequences, are in fact the sequence of iterations and therefore the computation of the weights at each update by the backpropagation algorithm.

dencies of the unknown variables (weights parameters) that the original neural network must tune. Consequently, different techniques can be modeled, including independent sub-network training and federated learning methods. Specifically, this mathematical model is based on a fixed-point function defined on an extended product space, see [6]: the sequences generated by the extended fixed-point function describe the aggregation of iterations produced by gradient descent algorithms applied to neural sub-networks. Using the fixed point theorem, it is possible to study the numerical convergence of sub-network-based approaches running back-propagation algorithms to the actual solution that would be obtained by original neural network.
The numerical convergence of sub-network partitioning algorithms in asynchronous mode is then examined. To this aim, we apply a classical convergence theorem of asynchronous iterations developed in [7], and in other papers such as [8], [9], to the aforementioned fixed-point function. The convergence conditions and cautious implementation details are then specified.

The rest of the paper is organized as follows: The related work to this paper is presented in Section 2. Section 3 recalls preliminaries on successive approximations, fixed-point principle and their usefulness to modelize sequential, synchronous and asynchronous iterations. The general mathematical model and the convergence study of the algorithms proposed by this paper are presented in Section 4. We first introduce the fixed-point mapping that models the behavior of a certain number of backpropagations on sub-networks of the original network. Then we prove thanks to proposition 1 and 2 the convergence result of synchronous and asynchronous sub-networks training algorithms. Section 5 is devoted to the empirical validation of the results, the experimental use case is also presented. Section 6 discusses this paper's results and presents future work and perspectives and section 7 concludes the paper.

2 Related Work

Stochastic gradient descent (SGD) uses a randomly chosen sample to update model parameters, providing benefits such as quick convergence and effectiveness. Nodes must communicate to update parameters during gradient calculation. However, waiting for all nodes to finish before updating can decrease computing speed if there is a significant speed difference between nodes. If the speed difference is small, synchronization algorithms work well, but if the difference is significant, the slowest node will reduce overall computing speed, negatively impacting performance. Asynchronous algorithms most effectively implement it [10]. Google's DistBelief system is an early implementation of async SGD. The DistBelief model allows for model parallelism [11]. In other words, different components of deep neural networks are processed by multiple computer processors. Each component is processed on a separate computer, and the DistBelief software combines the results. It is similar to how a computer's graphics card can be used to perform certain computations faster than the CPU alone. The processing time is decreased because each machine does not have to wait its turn to perform a portion of the computation, but can instead work on it simultaneously. Async SGD has two main advantages: (1) First, it has the potential to achieve higher throughput across the distributed system: workers can spend more time performing useful computations instead of waiting for the parameter averaging step to complete. (2) Workers can integrate information from other workers (parameter updates), which is faster than using synchronization (every N steps). However, a consequence of introducing asynchronous updates to the parameter vector, is a new issue known as the stale gradient problem. A naive implementation of async SGD would result in a high gradient staleness value. For example, Gupta et al. [12] show that the stale value of the average gradient is equal to the number of workers. If there are M workers, these gradients are M steps late when applied to the global parameter vector. The real-world impact is that high gradient staleness values can greatly slow down network convergence or even prevent convergence for some configurations altogether. Earlier async SGD implementations (such as Google's DistBelief system) did not consider this effect. Some ways to handle expired gradients include, among others, implementing “soft" synchronization rules such as in [13] and delaying the faster learner if necessary to ensure that the overall maximum delay is less than a threshold [14]. Researchers have proposed several schemes to parallelize stochastic gradient descent, but many of these computations are typically implemented using locks, leading to performance degradation. The authors of [15] propose a straightforward technique called Hogwild! to remove locks. Hogwild! is a scheme that enables locking-free parallel SGD implementations. Individual processors are granted equal access to shared memory and can update memory components at will using this method. Such a pattern of updates leads to high instruction-level parallelism on modern processors, resulting in significant performance gains over lock-based implementations. Hogwild! has been extended by the Dogwild! model in [16]. Dogwild! has extended the Hogwild! model to distributed-memory systems, where it still converges for deep learning problems. To reduce the interference effect of overwriting w at each step, the gradient ∇w from the training agents is transferred in place of w. The empirical evaluation of this paper is inspired by the Independent sub-network Training (IST) model [5], which decomposes the neural network layers into a collection of subnets for the same goal by splitting the neurons across multiple locations. Before synchronization, each of these subnets is trained for one or more local stochastic gradient descent (SGD) rounds. IST is better than model parallelism approaches in some ways. Given that subnets are trained independently during local updates, there is no need to synchronize them with each other. Still, IST has some of the same benefits as model parallelism methods. Since each machine only gets a small piece of the whole model, IST can be used to train models that are too big to fit in the RAM of a node or a device. This can be helpful when training large models on GPUs, which have less memory than CPUs. The following sections provide proof of convergence for partitioning-based networks in synchronous and asynchronous modes. The empirical evaluation section considers the IST model as a use case and proposes an asynchronous implementation.

3 Preliminaries: successive approximations. Synchronous and asynchronous algorithms

In the sequel, x_i^k denotes the k^{th} iterate of the i^{th} component of a multi-dimensional vector x.
A Numerical iterative method is usually described by a fixed-point application, say T. For example, given an optimization problem $\min_x f(x)$, the first challenge is to build an adequate T, such that the iterative sequence $x^{k+1} = T(x^k)$ for $k \in \mathbf{N}$ converges to a fixed point x^*, solution of $\min_x f(x)$. Successive iterations generated by the numerical method are then described by:

$$\begin{array}{l} Given\ x^0 = (x_1^0, ..., x_n^0)^T \in \mathbb{R}^n \\ \ for\ \ i = 1, ..., n \\ \quad for\ \ k = 1, 2...\ \ until\ convergence \\ \quad x_i^{k+1} = T_i(x_1^k, ..., x_n^k) \end{array} \tag{1}$$

Suppose that x is partitioned into m blocks $x_l \in \mathbb{R}^{n_l}$ so that $\Sigma_{l=1}^m n_l = n$ and that m processors are respectively in charge of a block x_l of components of x, then let us then define sets $S(k)$ describing the blocks of x updated at the iteration k. The other blocks are supposed not to be updated at iteration k. Asynchronous algorithms, in which the communications between blocks are free with no synchronization between the processors in charge of the blocks, are described by the following iterations:

$$\begin{array}{l} Given\ x^0 = (x_1^0, ..., x_m^0)^T \in \Pi_{l=1}^m \mathbb{R}^{n_l} \\ \ for\ \ k = 1, 2...\ \ until\ \ convergence \\ \quad for\ \ l = 1, ..., m \\ \quad x_l^{k+1} = \begin{cases} T_l(x_1^{\rho_1(k)}, ..., x_m^{\rho_m(k)})\ \ if\ \ l \in S(k) \\ x_l^k\ \ if\ \ l \notin S(k) \end{cases} \end{array} \tag{2}$$

Here $\rho_i(k)$ is the delay due to processor i at the k^{th} iteration. Note that with this latter formulation, synchronous per-blocks iterative algorithms are particular cases of asynchronous per-blocks iterative algorithms. Indeed, one has to simply assume that there is no delay between the processors, i.e., $\forall i, k\ \ \rho_i(k) = k$, and that all the blocks are updated at each iteration, $\forall k, S(k) = \{1, ..., m\}$. In this paper, $T^{(\beta)}$ will denote the fixed-point mapping describing the gradient descent method or its practical version SGD with learning rate β. Under suitable conditions and construction, the fixed-point application T describing the gradient descent method converges to a desired solution. In what follows, a fixed-point mapping $\mathcal{T}$ will also be introduced, this is a key function of our study on traininig deep neural netwoks. This function involves the computation of weighted parameters associated with network training, as well as combination matrices that model the aggregation of partial computations from sub-networks.

4 Convergence study

4.1 Mathematical modeling

Consider the general case of a feedforward neural network of some number of layers and neurons. Suppose that the number of interconnections between neurons in the network is n, so that the network is tuned by n weight vectors.
Let us consider a training set composed of inputs and targets. Consider a loss-function f (e.g. Mean Squared Error, Cross Entropy, etc.) that should be minimized on the traing set in order to fit the inputs to their targets. To do so, let $x \in \mathbb{R}^n$ be the weight vector that the neural network must tune to minimize f. Suppose that the gradient descent algorithm is used to perform this minimization. The gradient descent algorithm performs the iterations:

$$x^{k+1} = x^k - \beta \nabla_x f(x^k), k \in \mathbb{N}, \quad x^0 \in D(f) \tag{3}$$

Under the right conditions on the function f and the learning rate β, the function $I - \nabla_x f$ is contractive on its definition domain $D(f)$ and the successive iterations (Eq-3) converge to x^* satisfying the solution:

$$x^* = argmin_x f(x), \quad x^* \in D(f) \tag{4}$$

Here $\nabla_x f(x) = (\frac{\partial f_i}{\partial x_j})_{ij}$ is the Jacobian matrix consisting of the partial derivatives of f with respect to weight vector x of the interconnections of the network. A Backpropagation with a learning rate β_i applied to the network consists in computing successive iterations associated to the Descent gradient algorithm:

$$\begin{aligned} &x^0 \ \textit{being the initialization weights} \\ &x^{k+1} = x^k - \beta_i \nabla_x f(x^k), \quad k \in \mathbb{N}, \end{aligned} \tag{5}$$

Let $T^{(\beta_i)}$ denotes the fixed point application describing the iterations (Eq-5):

$$y = T^{(\beta_i)}(x) = x - \beta_i \nabla_x f(x)$$

Consider m backpropagation algorithms on the neural network Net with learning rates β_i, $i = 1, ..., m$. The learning rates $\beta_1, ..., \beta_m$ are chosen in such a way that the gradient descent algorithms converge to x^*, solution of (4), i.e.

$$T^{(\beta_i)}(x^*) = x^*, \forall i \in \{1, ..., m\} \tag{6}$$

For a network Net with n weighted vector, and m backpropagation algorithms, let us introduce the following fixed-point mapping which is called aggregation fixed point mapping:

$$\begin{aligned} \mathcal{T} : (\mathbb{R}^n)^m &\longrightarrow (\mathbb{R}^n)^m \\ (x^1, ..., x^m) &\longrightarrow (y^1, ..., y^m) \\ y^l &= T^{(\beta_l)} \left(\sum_{k=1}^{m} E_k x^k \right) \end{aligned} \tag{7}$$

where E_k are diagonal matrices called weight matrices so that

$$\sum_{k=1}^{m} E_k = I \tag{8}$$

I is the identity matrix $\in \mathbb{R}^{n \times n}$. Since the weight matrices are diagonal, the aggregation fixed-point mapping modelizes the execution of m backpropagations on m sub-networks. It should be noted that condition (8) is essential for the formal convergence study.

Proposition 1. ***$\mathcal{T}$ is a contractive function***
The aggregation fixed-point mapping $\mathcal{T}$ is contractive with constant of contraction α satisfying, $\alpha \leq \max_{l=1}^{m}(\alpha_l)$, where α_l are the constants of contraction of $T^{(\beta_l)}$

Proof 1 *Let $y^l = T^{(\beta_l)}(\Sigma_{k=1}^{m} E_k x^k)$*
From (6) each gradient descent converges to x^* the solution of (4), so each fixed-point mapping $I - \beta\nabla_x J$ is contractive.
For $l \in \{1, ..., m\}$, denote by α_l the constant of contraction of $T^{(\beta_l)}$ with respect to the l_2 norm $\|x\|_2$.
By (7) we have

$$\|y^l - x^*\|_2 = \|T^{(\beta_l)}(\Sigma_{k=1}^{m} E_k x^k) - T^{(\beta_l)}(x^*)\|_2 \leq \ \alpha_l \|\Sigma_{k=1}^{m} E_k x^k - x^*\|_2$$

Since E_k are diagonal matrices and $\Sigma_{k=1}^{m} E_k = I$, we have

$$\|\Sigma_{k=1}^{m}(E_k x^k - x^*)\|_2 = \|\Sigma_{k=1}^{m} E_k(x^k - x^*)\|_2 \leq \max_{k=1}^{m} \|x^k - x^*\|_2$$

Thus, $\|y^l - x^*\|_2 \leq \alpha_l \ \max_{k=1}^{m} \|x^k - x^*\|_2$
Hence we obtain, $\max_{l=1}^{m} \|y^l - x^*\|_2 \leq (\max_{l=1}^{m} \alpha_l) \ \max_{k=1}^{m} \|x^k - x^*\|_2$
Let's define the norm, $\|(y^1, ..., y^m)\|_\infty = \max_{1 \leq l \leq m} \|y^l\|_2$, then

$$\|\mathcal{T}\left(x^1, ..., x^m\right) - (x^*, ..., x^*)\|_\infty \leq (\max_{l=1}^{m} \alpha_l)\| \left(x^1, ..., x^m\right) - (x^*, ..., x^*)\, \|_\infty$$

This proves the claimed result.

Proposition 2. ***Asynchronous convergence of $\mathcal{T}$***
The asynchronous iterations generated by the fixed-point mapping $\mathcal{T}$ converge to $(x^*, ..., x^*)^T$.

Proof 2 *Proposition 1 implies that we are in the framework of contractive fixed-point mapping with respect to a maximum norm on a product space. Indeed, $\mathcal{T}$ is a contractive mapping with respect to the maximum norm $\|..\|_\infty$, on the product space $\Pi_{i=1}^{m}\mathbb{R}^n = (\mathbb{R}^n)^m$. These are the required sufficient conditions for the convergence of asynchronous algorithms associated to $\mathcal{T}$. see [7]. Thus Proposition 2 is proved.*

Remark 1 *The aggregation fixed point mapping modelizes Independent Sub-network Training, indeed, if the neurons are disjoint then $\forall i, (E_k)_{ii}$ is equal to either 0 or 1. It should be noted that we don't use the notion of compressed iterates and the assumptions related to the expectancy of the compression operator $\mathcal{M}(.)$ as in [5]. We prove the convergence of the iterates values instead of their convergence on expectation. To modelize federated learning like algorithms the weighted matrices become $\forall i, (E_k)_{ii} = 1/M$.*

5 Empirical validation

5.1 The links between the mathematical model and the implementation

- To obtain asynchronous iterations produced by the aggregation of the computations of m sub-networks, it is sufficient to consider that $T = \mathcal{T}$ in (2).
- Synchronous per-blocks computations correspond to $S(k) = \{1, ..., m\}$, and ${\rho_i}^k = k$, $\forall k$ (no-delays and all the blocks are updated at each iteration).
- A sub-network l is defined by all nodes j corresponding to the non-zero entries of E_l: $(E_l)_{jj} \neq 0$, so the aggregation of the weight vectors computed by the sub-networks must be done carefully so that $\sum_{k=1}^{m} E_k = I$ is satisfied.
- Even if in the mathematical formulation $\mathcal{T}$ is defined on the product space $\Pi_{i=1}^{m} \mathbb{R}^n$, each sub-network l updates at each iteration only its own components, this is because the weight matrices E_l are diagonal matrices and $(E_l)_{jj} = 0$ if node j does not belongs into sub-network l.
- To execute the computations in asynchronous mode, it should be noted that the convergence theory requires some conditions on the block-component updates and the delays between the processors. These are listed below.

- $\forall i \in \{1, ...m\}$ the set $\{k \in \mathbb{N} / i \in S(k)\}$ is infinite. This simply means that any sub-network i is guaranteed to update its computations, i.e. not to be permanently inactive.
- The delays must "follow" the iterations, the mathematical formulation for that is $\forall i, \lim_{k \to \infty}(\rho_i(k)) = \infty$.

For more precision see the literature on asynchronous iterations, e.g. [7], [17], [18], [9], [19],

It can be noticed that in a practical implementation these assumptions are realistic.

5.2 Use case: Independent Subnet Training

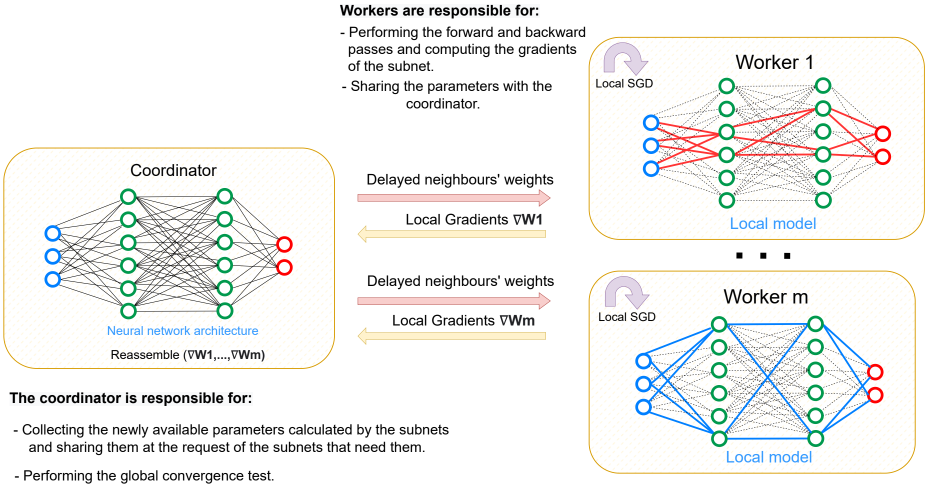
As a use case, let us consider the IST model for this research. Note that the proposed mathematical model is general and could also cover dependent partitioning-based networks. The publicly available Github implementation[3], the clarity of

[3] https://github.com/BinhangYuan/IST_Release

the code, and the recent date of publication were among the reasons to consider this model for our use case.

In the original work, the authors suggest to regularly resample the subnets and to train them for fewer iterations between resamplings. This paper proposes that the coordinator distribute subnets only at the beginning of training to prevent blockage. Similarly to the original IST model, the following constraints are respected:

- Input and output neurons are common to all subnets.
- Hidden neurons are randomly partitioned via uniform assignment to one of n possible compute nodes.
- The complete Neural Network (NN)'s weights are partitioned based on the number of activated neurons in each subnet.
- No collisions occur because the parameter partition is disjoint.



Workers are responsible for:

- Performing the forward and backward passes and computing the gradients of the subnet.
- Sharing the parameters with the coordinator.

The coordinator is responsible for:

- Collecting the newly available parameters calculated by the subnets and sharing them at the request of the subnets that need them.
- Performing the global convergence test.

Fig. 1: The architecture of a distributed asynchronous training based on the partitioning-based framework.

The training process of the asynchronous IST training is as follows: (1) The coordinator distributes the sub-networks to the workers at the beginning of the training process. Contrary to the original IST, no new group of subnets is constructed to avoid synchronization. Note that during the training process, the coordinator trains a subnet, not the full neural network. (2) Each cluster (coordinator and workers) computes the loss of its local model based on its local data (i.e., forward pass). (3) Each cluster computes the gradients based on the loss (i.e., backward pass). (4) Each worker sends its parameters to the coordinator. In our asynchronous version, it is done using Pytorch's *isend* method that sends a tensor asynchronously. (5) Given that the partition is disjoint, the coordinator copies the parameters into the full neural network without collisions. The asynchronous version is considered using Pytorch's *irecv* method that receives a tensor asynchronously.

The architecture and training procedure considered in this research (Figure 1) are comparable to frameworks such as the parameter server with centralized or federated learning techniques [20]. The local models, however, are subnets of the full model and not a replica. Note that Figure 1 illustrates a general partitioning-based framework described by the proposed mathematical model and not necessarily the particular IST model. To represent the async IST implemented by this paper, one could ignore the transmission of the delayed parameters from the coordinator to the workers after the first partition, given that each worker has an independent subnet that does not require the parameters of other workers. One of the challenges of asynchronous training is that the asynchronous communication will be rendered meaningless if one continues to allow communication between workers: This issue is solved by avoiding parameters exchange between subnets and avoiding synchronization during local updates or the coordinator's updates.

5.3 Experiments

The Google Speech [21] and Mafaulda[4] [22] datasets were considered to demonstrate the advantage of asynchronous distributed training of neural networks over synchronous training. The Google Speech dataset describes an audio dataset of spoken words that can be used to train and evaluate keyword detection systems. The objective is to classify 35 labeled keywords extracted from audio waveforms. In the dataset, the training set contains roughly 76,000 waveforms, and the testing set has around 19,000 waveforms. The Mafaulda dataset is a publicly available dataset of vibration signals acquired from four different industrial sensors under normal and faulty conditions. This database is composed of 1951 multivariate time-series and comprises six different states. Similarly to [5], a three-layer MLP (multilayer perceptron) has been developed. The number of neurons in each layer is a parameter. Using subnets as mentioned in [5], the MLP is distributed across all processes and GPUs. To simplify, links between neurons are divided among all the nodes, and each link is controlled exclusively by a single node. Consequently, the coordinator node (which also participates in the training computation) and the worker node share link weights during parallel training.

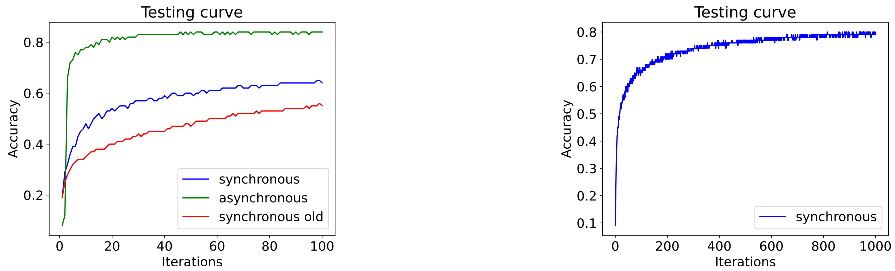
In contrast to [5], where all tasks are synchronized, an implementation with asynchronous tasks and iterations has been built. In this version, there is no synchronization following the initialization. The coordinator node has a heavier workload compared to other nodes in the implementation of the IST code. It is responsible for coordinating workers, performing tests with the test dataset,

[4] Dataset: https://www02.smt.ufrj.br/~offshore/mfs/page_01.html, last visited: 20-10-2022

sending and receiving the link weights of other nodes' neurons. Additionally, the coordinator is also responsible for a subnet (local gradient descent) in this implementation, leading to a higher amount of calculations.

For the experiments, the French supercomputer Jean Zay of GENCI was used. This supercomputer contains more than 2,000 GPUs (most of them are NVIDIA V100).

The experiments compare the performance of three different versions of a neural network training algorithm: the old synchronous version (the training version implemented by [5]), the synchronous version with partitioning at the first iteration, and the asynchronous version with no synchronizations between the workers. The main point that the experiments aim to prove is that the asynchronous method provides better testing accuracy in a shorter time and lower execution times compared to the synchronous methods.



(a) Testing curve for the synchronous and the asynchronous cases (limited to 100 epochs).

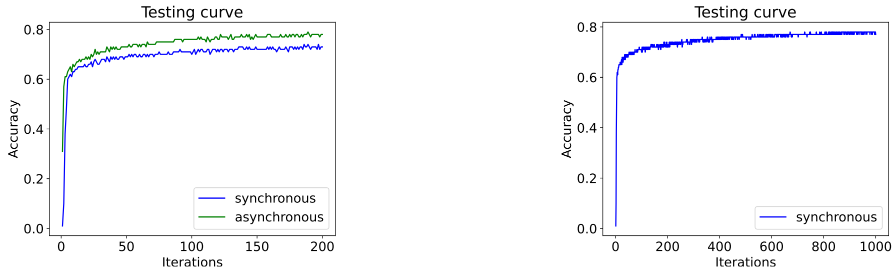
(b) Testing curve for the synchronous case (limited to 1,000 epochs).

Fig. 2: Experiments with the Google Speech dataset.

Table 1: Execution times for the Google Speech dataset.

Algorithm	Number of epochs	Execution times (in s)	Accuracy	Asynchronous acceleration
Synchronous old version [5]	100	531	0.56	1.08
Synchronous version	100	550	0.67	1.11
Synchronous version	1,000	5,501	0.80	11.18
Asynchronous version	100	492	0.84	1

The training of three models was executed ten times with 64 GPUs. A supercomputer with specialized architecture was used, resulting in minimal variations in execution times. Testing accuracy was averaged and plotted to assess the convergence of the models. Results from Figures 2a and 3a indicate that the asynchronous method outperforms the synchronous mode due to worker nodes completing more iterations than the coordinator node (workers don't wait for the coordinator to synchronize at the end of each epoch), while in synchronous mode, the number of iterations completed is equal to the number of epochs. The number of epochs was fixed at 100 for the Google Speech dataset and 200 for Mafaulda based on several experiments, as those values better show the difference in performance between synchronous and asynchronous training. Additionally, the average testing accuracy for 1,000 epochs was computed only for the synchronous model with one partitioning at the beginning to compare the performance of the asynchronous model with a smaller number of epochs against the synchronous model's accuracy after 1,000 epochs. Figures 2b and 3b demonstrate that the synchronous case has lower testing accuracy even with 1,000 epochs compared to the asynchronous case. Table 1 shows the average execution times rounded to the nearest second of three versions. The maximum accuracy is shown. The last column shows the asynchronous acceleration compared to the other versions. The asynchronous training was faster than the other synchronous approaches on both Google Speech and Mafaulda datasets. The code used for the experiments is available[5].



(a) Testing curve for the synchronous and the asynchronous cases (limited to 200 epochs).

(b) Testing curve for the synchronous case (limited to 1,000 epochs).

Fig. 3: Experiments with the Mafaulda dataset.

6 Discussion and Perspectives

This paper aims to address numerical convergence in deep learning frameworks for large problems. To achieve this, the neural network is partitioned into subnets and back-propagation algorithms are applied to these subnets instead of the original network. The paper also raises questions about reconstituting the

[5] https://github.com/rcouturier/async_mlp

global solution from partial solutions of subnets and the differences between synchronous and asynchronous algorithms during convergence.

The paper presents a mathematical model that describes the behavior of partitioning-based network methods in synchronous and asynchronous modes, and proves the convergence of such methods. To validate this approach, the specific case of Independent Subnets Training is considered, and the asynchronous mode is implemented and compared to the standard synchronous mode. The study concludes that the asynchronous mode has better performance in terms of execution time and accuracy.

The presented mathematical model is general and describes various scenarios, including federated learning with workers possessing the same network architecture. Future works will investigate different concepts and distributed architectures to broaden the scope of the study.

7 Conclusion

This paper proposes a general model to investigate the convergence of distributed asynchronous training of deep neural networks. The model addresses the question of whether partial solutions of sub-networks can be used to reconstitute the global solution. Two experimentations based on the Independent sub-network Training model on different datasets are provided, demonstrating that using the asynchronous aggregation sub-networks model is interesting for reducing execution time and increasing accuracy, even on a supercomputer. The study expects a better performance from asynchronous aggregation algorithms in contexts with weak computation time to communication time ratios.

Acknowledgment

This work was partially supported by the EIPHI Graduate School (contract "ANR-17-EURE-0002"). This work was granted access to the AI resources of CINES under the allocation AD010613582 made by GENCI and also from the Mesocentre of Franche-Comté.

References

1. T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.
2. J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *Acm computing surveys (csur)*, vol. 53, no. 2, pp. 1–33, 2020.
3. S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

4. D. Podareanu, V. Codreanu, T. Sandra Aigner, G. C. van Leeuwen, and V. Weinberg, "Best practice guide-deep learning," *Partnership for Advanced Computing in Europe (PRACE), Tech. Rep*, vol. 2, 2019.
5. B. Yuan, C. R. Wolfe, C. Dun, Y. Tang, A. Kyrillidis, and C. Jermaine, "Distributed learning of fully connected neural networks using independent subnet training," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1581–1590, 2022.
6. J. Bahi, J.-C. Miellou, and K. Rhofir, "Asynchronous multisplitting methods for nonlinear fixed point problems," *Numerical Algorithms*, vol. 15, no. 3, pp. 315–345, 1997.
7. M. El Tarazi, Nabih, "Some convergence results for asynchronous algorithms," *Numerische Mathematik*, vol. 39, no. 3, pp. 325–340, 1982.
8. G. Baudet, "Asynchronous iterative methods for multiprocessors," *J. Assoc. Comput. Mach.*, vol. 25, pp. 226—-244, 1978.
9. J. Bahi, "Asynchronous iterative algorithms for nonexpansive linear systems," *Journal of Parallel and Distributed Computing*, vol. 60, no. 1, pp. 92–112, 2000.
10. X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *International Conference on Machine Learning.* PMLR, 2018, pp. 3043–3052.
11. J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.
12. S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *2016 IEEE 16th International Conference on Data Mining (ICDM).* IEEE, 2016, pp. 171–180.
13. W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.
14. Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," *Advances in neural information processing systems*, vol. 26, 2013.
15. B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in neural information processing systems*, vol. 24, 2011.
16. C. Noel and S. Osindero, "Dogwild!-distributed hogwild for cpu & gpu," in *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014, pp. 693–701.
17. D. P. Bertsekas and J. N. Tsitsiklis, "Parallel and distributed computation: numerical methods," 2003.
18. P. Spiteri, "Parallel asynchronous algorithms: A survey," *Advances in Engineering Software*, vol. 149, p. 102896, 2020.
19. A. Frommer and D. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1, 2, pp. 201–216, 2000.
20. A. M. Elbir, S. Coleri, and K. V. Mishra, "Hybrid federated and centralized learning," in *2021 29th European Signal Processing Conference (EUSIPCO).* IEEE, 2021, pp. 1541–1545.
21. P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018. [Online]. Available: https://arxiv.org/abs/1804.03209
22. F. M. Ribeiro, M. A. Marins, S. L. Netto, and E. A. da Silva, "Rotating machinery fault diagnosis using similarity-based models," *XXXV Simpósio Brasileiro de Telecomunicações e Processamento de Sinais-sbrt2017*, 2017.