Technical University of Crete

School of Electrical & Computer Engineering

Diploma Thesis

Federated Learning at TensorFlow Using the Geometric Approach

A Thesis Submitted in Partial Fulfillment of the Requirements for the Undergraduate Degree in Electrical and Computer Engineering

Author:

Georgios Frangias

Thesis Committee:

Prof. Antonios Deligiannakis (*Supervisor*)

Asst. Prof. Nikos Giatrakos

Assoc. Prof. Vasilis Samoladas

December 2023

Abstract

The rapid growth of data generation and internet usage in recent years has created an unprecedented demand for efficient Big Data collection, processing and analysis. The ever-growing privacy concerns of the public opinion and the enactment of regulations on this subject, induce the need for the development of decentralized, distributed and scalable Machine Learning mechanisms, that can assure both personal data security and high accuracy collective training. The scientific field of Federated Learning is dedicated to achieving exactly that; train a global machine learning model without communicating sensitive locally generated data. For the purpose of the current thesis, we have developed a deployable extension to the Distributed Machine Learning library KungFu, to effortlessly execute Federated Learning training jobs on decentralized compute nodes. The implemented algorithms are the three Functional Dynamic Averaging methods, inspired by the Geometric Approach. These algorithms have the ability to approximately monitor a global threshold function, using solely local data and, subsequently, dynamically determine the need for synchronization and model aggregation. We have put our implementation to the test by executing exhaustive experiments on multi-node GPU infrastructure, and compared it to a classic distributed algorithm. The results demonstrate a significant training time reduction, due to reduced communication overhead, without having repercussions on accuracy, especially for non-ideal network topologies.

Acknowledgements

First of all, I would like to express my deep gratitude to my supervisor, Professor Antonios Deligiannakis. Without his support and knowledge on the subject, this work would not have been possible. His insights in his courses related to programming systems, databases and sensor networks provided me with the fundamental thought process necessary to accomplish this achievement. His passion for research was not only inspiring to me but also contagious, encouraging me to delve deeper into the field.

My gratitude extends to the committee members. Professor Vasilis Samoladas was always available for technical questions and assistance. His teaching during the course on distributed systems introduced me to the fascinating concepts of this field. Furthermore, I would like to thank Professor Nikos Giatrakos for his participation in the thesis committee, his commitment to research and his interest in the work I have done.

I would also like to acknowledge the support I have been given by the research team whom I was part of at the Technical University of Crete. Our team consisted of faculty members and fellow students. It was always helpful to discuss the technicalities and next steps of our work, with each of us contributing to our collective knowledge.

Last but not least, I am truly grateful to my family and friends who supported me throughout my five-year-long undergraduate studies. Without them, this journey would have seemed much more difficult or even impossible.

Contents

List of Figures

Chapter 1

Introduction

As of 2018 it is estimated that 2.5 million terabytes of data is created daily and in 2023 around 65% of the world population is frequently using the internet. In this era characterized by an overwhelming abundance of data, the need for efficient data collection, analysis, and categorization has never been greater. Machine Learning (ML) has not only played a pivotal role in meeting this demand but is also shaping the technological landscape for the foreseeable future. Its applications are vast and impactful, spanning from large language models and image recognition to online recommendation systems, spam detection, and even the creation of digital art. As a result, the scientific community is relentlessly engaged in advancing ML technologies through novel techniques.

Traditional ML techniques might not be suitable while dealing with large amounts of data. The computational complexity is high and thus the parallelization of the learning process is necessary. An adaptable and scalable network of learners is required, with an emphasis on edge computing for efficient learning allocation. Minimizing data exchange is crucial to optimize network efficiency and resource utilization, all while maintaining the privacy for individual users. These challenges underscore the importance of distributed and federated learning approaches, which are designed to address these requirements.

1.1 Background

To fully grasp the focus of this thesis, it's crucial to first understand the core concepts of Machine Learning (ML), Distributed Machine Learning (DML) and Federated Learning (FL).

ML is a specialized area within the field of Artificial Intelligence (AI) dedicated to enabling computers to learn from data in order to make informed decisions or forecasts. In contrast to conventional computational approaches that use fixed human-coded algorithms to compute and solve problems, ML algorithms have the ability to recognize patterns in data and make inferences accordingly. This way, computers gradually learn and improve their performance as they are exposed to more data.

Most ML algorithms are fundamentally dependent on the concept of Artificial Neural Networks (ANNs), which are modelled after the biological neural networks that constitute the brain. Similar to their biological counterparts, ANNs consist of several connected

processors called neurons that each produces a sequence of real-valued activations. An ANN has an input layer of neurons that get activated directly by the input data, as well as additional neurons that get activated indirectly through weighted connections to other previously activated neurons. These weights imply a connection strength between neurons across different layers, akin to synaptic connections in the human brain. Learning in an ANN setup is the process at which the optimal ANN variables —such as weights— and subsequent behavior are found.

Traditional non-distributed ML is performed centrally on a single machine. The data is collected and preprocessed on the same machine, and the model is trained using this data, adjusting its variables iteratively, to minimize a specific loss function. This loss function serves as a metric, operating in the domain of the model's [1] variables, to approximate how closely the model's predictions align with the actual characteristics of the data.

In various scenarios, a distributed learning system is advantageous, primarily because it can significantly accelerate the learning process through high-level parallelization and can prevent the communication of sensitive data over the network to a central server. This is where Distributed Machine Learning (DML) comes into play, offering enhancements in performance, accuracy, and the capacity to handle larger data volumes. Within DML systems, multiple nodes, and often referred to as clients, maintain their own models and independently conduct training on subsets of the complete dataset. It's crucial that the dataset is distributed in an independent and identically distributed (IID) manner to ensure the effectiveness of DML algorithms. The training unfolds iteratively in cycles known as communication rounds. At the conclusion of each communication round, the clients transmit only their model variables across the network for aggregation. Subsequently, the aggregated model is transmitted back to the clients, initiating a new learning cycle, i.e., communication round.

While DML offers substantial performance gains over traditional, non-distributed ML methods, it is confined to IID datasets, which are quite rare in real-world learning scenarios. This mostly ends up with data being centrally collected in a cluster-based server, so that they get distributed accordingly. So, in most scenarios DML faces privacy challenges. The advent of data protection regulations like the EU's General Data Protection Regulation (GDPR) and the public's concerns on personal data after major global data breaches necessitate a more privacy-conscious approach to learning from multiple remote users. Federated Learning (FL) addresses this need by providing a distributed learning framework that prioritizes user privacy. Instead of collecting all personal data from local devices to a centralized cluster server, FL performs the training locally in the edge devices. Only the model variables, not the sensitive personal data, are then aggregated centrally. This focus extends not only to enhancing data privacy but also to reducing network overhead. Since the raw data isn't transferred across the network, FL mini-

[1] It is important to note that although ANNs are the most common ML models they are not the only ones. A ML model is any kind of algorithm used to find patterns and classify data. Notable non-ANN models include Support Vector Machines, Bayesian Classifiers and the k-nearest neighbor algorithm. In this thesis, the term "model" is used interchangeably with ANNs; however, it's important to recognize that ANNs are just one type of ML model.

mizes the burden on network resources, giving it a significant advantage over traditional DML techniques. This is especially important for large scale learning implementations involving millions of users that lack the network capabilities of data centers.

The big technology company Google has been at the forefront of promoting and implementing FL since 2016. They view FL playing a crucial role in Distributed Machine Learning, especially for small mobile devices —like smartphones— and have already implemented it on Gboard, Google's attempt on a virtual keyboard. They have developed TensorFlow [25] in 2015, which is an open-source ML library that provides templates for neural networks, data flow graphs and ML algorithms. TensorFlow offers certain strategies for distributed learning, such as the Multi-Worker Mirrored Strategy, aimed at facilitating distributed system operations. However, as of now, TensorFlow's offerings for Federated Learning (FL) are limited. The TensorFlow Federated (TFF) framework exists, but is purely simulation-based and not deployable. Despite this framework's presence, TensorFlow has yet to provide an integrated solution that incorporates FL algorithms.

The most prominent framework for Distributed Machine Learning (DML) that utilizes TensorFlow, along with other libraries like PyTorch, is Horovod, initially developed by Uber [34]. Horovod facilitates inter-GPU communication and achieves high throughput speeds through the use of ring reduction techniques. Available under the Apache 2.0 license, it allows for real-world, deployable distributed learning experiments with minimal code modification. In a similar vein, KungFu by Luo Mai et al. [7, 24] employs adaptive distributed learning policies to dynamically adjust learning hyper-parameters. This allows it to achieve superior data throughput, irrespective of the network limitations faced by the learner nodes. KungFu also offers tools for monitoring the learning process, making it a comprehensive solution for distributed machine learning. This adaptability gives KungFu a performance edge over Horovod, particularly in scenarios involving a large number of nodes.

1.2 Problem Statement

Most existing distributed and federated learning algorithms in use set fixed communication rounds, often lasting for a single training step. This communication strategy can be quite trivial and inconsequential for a cluster-based DML environment, but it is restrictive for networks of distant learners with limited internet connection. The communication cost becomes enormous in relation to the computational cost of training, leading the training process to succumb to communication overhead.

The Geometric Approach [35] method was one of the pioneers in functional monitoring over distributed systems. It paved the way to monitor specific global metrics, by locally computing estimations, with minimal network communication. Functional Dynamic Averaging (FDA) [33] has been developed, influenced by the Geometric Approach, and it is designed with DML in mind. It has introduced three methods to approximate the variance between local models and their average model, and it has built a synchronization strategy around them. The FDA methods are theoretically proven to significantly reduce communication cost and total training time.

In this thesis, our goal is to implement and test Geometric Approach techniques in a deployable form using the TensorFlow library. This includes a deep dive into the fundamentals of Optimization, Machine Learning, Distributed Machine Learning and Federated Learning. By this we aim to achieve substantial communication overhead reduction, an important premise in Federated Learning.

1.3 Thesis Outline

This thesis statement consists of six chapters. Excluding the current one, these are;

2. Related Work
   In this chapter, we lay the foundational work for Artificial Neural Networks, Optimizers, Distributed Machine Learning and Federated Learning. These concepts are explained in detail, organized by level of complexity, starting from the simplest.

3. Selected Tools and Framework
   We evaluate the available software tools relevant to the subject and analyze the reasoning behind the selection process.

4. Methodology
   A thorough analysis on the Geometric Approach and Functional Dynamic Averaging is conducted. Moreover, we elaborate on the adaptations needed to incorporate these algorithms in the tools we have used.

5. Experiments and Results
   The executed experiments and their logic are explained. Experiments results are presented and investigated.

6. Conclusions
   We conclude analyzing the experiments results and research process, and suggesting further work on the subject.

Chapter 2

Related Work

2.1 Artificial Neural Networks

Machine Learning research is fundamentally dependent on *Artificial Neural Networks (ANNs)*, a computational model that resembles the function of biological neural networks. The primary role of ANNs is to accurately classify incoming data, which can be of many forms, such as text, voice and images. To achieve this, ANNs undergo an iterative learning process, independently tweaking their internal parameters. If designed correctly, the network will eventually make predictions that are progressively more accurate, resembling human cognitive ability in certain tasks. Structurally, an ANN consists of several processing elements, called neurons or perceptrons, that receive input data and deliver outputs based on specific functions assigned to them. In this section, we will discuss the inner workings of ANNs in detail.

2.1.1 Perceptrons and Neural Network Architecture

For now, let's think of perceptrons as black boxes that receive inputs and produce an output. In an ANN perceptrons are organized into multiple layers, with the most basic ANNs consisting solely of an input layer and an output layer. The input layer simply receives raw data values, while the output layer produces the network's classification predictions. Correspondingly, the number of perceptrons in the input layer is determined by the dimensions of input data, and the number of perceptrons in the output layer matches the number of possible classes. After the learning process, the goal is for the ANN to classify incoming data with high confidence. Specifically, the perceptron in the output layer that corresponds to the correct class should produce a value close to 1, while values from other output perceptrons should remain close to 0.

In more complicated ANN architectures multiple layers are placed between input and output, called hidden layers. Multiple hidden layers can be useful for the neural network to recognize input data features in different levels of abstraction. Each perceptron of a layer is connected to all the perceptrons of the previous layer. The input of a perceptron that doesn't belong to the input layer is a weighted sum of the outputs of all perceptrons in the previous layer. The hidden layers have an arbitrary number of perceptrons and distinct functions.

In Figure 2.1 a simple ANN is presented. There are four layers of perceptrons. The perceptrons are represented by circles and each connection between neighboring layers has a weight value. The input layer receives raw data, in this example an image of a handwritten "3" digit. The output layer gives an estimation of the input's label [1]. In this example, as the network tries to classify handwritten digits the output layer is designed to have 10 perceptrons, as many as the number of single digits. In the output layer the first perceptron's value is the probability of the inputted image to be a 0, the second to be 1 and so on.

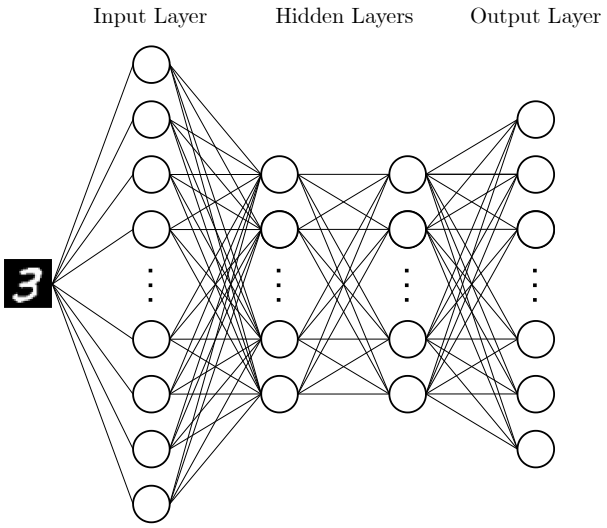


Figure 2.1: Artificial Neural Network

2.1.2 Forward Propagation

In the beginning, it's essential to note that all the weights of the neural network are initialized with random values. Additionally, the data intended for input to the neural network should undergo preprocessing, which typically includes normalization and standardization. The data is partitioned into two distinct sets: the training dataset and the test dataset. The training dataset serves as the source from which data is drawn to train the neural network, while the test dataset remains concealed from the neural network throughout the training process. The purpose of the test dataset is to assess the accuracy of the neural network's predictions when faced with new, unseen data.

The learning process is initiated by a process called *forward propagation*. At each forward propagation step a subset of the training dataset —called batch— is fed to the input layer of the neural network. Eventually, after traversing all layers one-by-one, eventually it outputs its predictions. The question is how single perceptrons derive their output value from the input.

As said before, each perceptron that doesn't belong to the input layer, receives as input, the outputs of the previous layer's perceptrons, along with the corresponding

[1]The terms label and class are used interchangeably and they express the category in which data items are assorted.

weights. The problem is, that the weighted sum of the inputs that a perceptron computes can span the entire set of real numbers. To confine the output to a more manageable range —typically between $[0, 1]$— this sum is passed through an *activation function*. Common properties of activation functions include non-linearity, differentiability, and monotonicity, along with a compact codomain. Widely used activation functions include the sigmoid function, hyperbolic tangent function (tanh) and rectified linear unit (ReLU). All of these functions are activating —each in their own way— their input around zero. However, there are instances where a perceptron needs to be activated around a different value. To accommodate this, many ANNs equip each perceptron with its own *bias* value, in addition to its weighted connections. This allows for more flexible activation behavior.

Let there be a simple architecture of two layers independent of their position in the ANN. The first layer has m perceptrons and the following second layer has n perceptrons. The output of the first layer is $\mathbf{x}$ with dimensions $(m \times 1)$, the output of the second layer is $\hat{\mathbf{y}}$ with dimensions $(n \times 1)$, the weights of the in between connections is given by the matrix $\mathbf{W}$ with dimensions $(m \times n)$, the bias of each perceptron is set by $\mathbf{b}$ with dimensions $(n \times 1)$ and $\sigma()$ is the activation function. The output of the perceptrons of the n-lengthed layer is calculated by the function,

$$\mathbf{u} = \mathbf{W}^T\mathbf{x} + \mathbf{b}$$

$$\hat{\mathbf{y}} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b}) = \sigma(\mathbf{u}). \tag{2.1}$$

Figure 2.2 shows this example in a system-like form where $\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_n$ are the rows of $\mathbf{W}$,

$$\begin{bmatrix} \hat{y_1} \\ \hat{y_2} \\ \vdots \\ \hat{y_n} \end{bmatrix} = \sigma \left(\begin{bmatrix} \mathbf{w}_1 \\ \mathbf{w}_2 \\ \vdots \\ \mathbf{w}_n \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right). \tag{2.2}$$

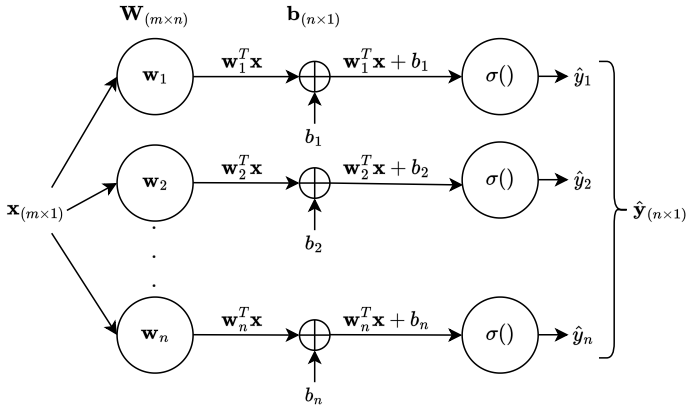


Figure 2.2: Input and output of a layer of perceptrons
Inspired by [18]

Now it is quite clear how the forward pass is done. A batch of data is inputted in

the input layer, many calculations involving weights and biases happen in hidden layers' perceptrons until an output is derived at the output layer. At the beginning of the learning process the weights and biases values are initialized randomly. Consequently, one could argue that initially, the neural network makes random guesses regarding the actual labels of the data. This leads to the question of how the ANN progressively improves its ability to label data with increased accuracy and confidence.

2.1.3 Loss Computation

To enhance the predictive ability of an ANN, a metric on the accuracy and confidence of the neural network is necessary. This metric is computed through the use of a *loss* or *cost function*, which takes into account the network's output and the values that the ANN would ideally produce. Sticking to the example that was used beforehand, if $\hat{\mathbf{y}}$ is the network's prediction and $\mathbf{y}$ is the ideal values, the loss function for one data item should be a function of the form $\ell(\mathbf{y}, \hat{\mathbf{y}})$. But as $\hat{\mathbf{y}}$ is completely dependent on the data item $\mathbf{x}$ (the ANN's input) as well as the variables of the model $\mathbf{W}$ and $\mathbf{b}$, the function can be represented as $\ell(\mathbf{W}, \mathbf{b}; \mathbf{x}, \mathbf{y})$.

The main objective of learning in an ANN setting is to minimize the value of the loss function. The variables of the function are the weights and biases of the ANN. It's a well-established principle that to approach the minimum, whether it's a local or global minimum, of a function, one must move in the trajectory opposite to the gradient of the function. This is analyzed in more detail in Sections 2.2 and 2.3. However, the problem that arises is, that the loss function is rendered practically unknown. The domain of the function can have thousands or millions of dimensions, due to the large number of perceptrons used, and the sheer volume of pre-labeled data that requires evaluation can further complicate the task of estimating it accurately.

2.1.4 Backpropagation

A technique called backpropagation is employed to make iterative estimations of the loss function's gradient and consequently adjust the weights of the neural network accordingly. The algorithm was independently invented by multiple researchers but it was popularized by D. Rumelhart, G. Hinton and R. Williams [31]. Backpropagation is employed after every forward pass and loss computation and estimates the gradient of the loss function layer-by-layer using the chain rule. Starting from the output layer and moving backwards, at each perceptron connection an error signal e is calculated and then used to update the corresponding weight value.

Coming back to the Equation 2.1 and Figure 2.2, let there be an ANN with k layers and $x_j^{(m-1)}$ be the j-th output of the $(m-1)$-th layer and therefore the j-th input to the perceptrons of the m-th layer [1]. Also, let the loss function $\ell(\mathbf{x}, \mathbf{y}; \mathbf{W})$ be a simple squared error function. The weighted sum of inputs at the i-th perceptron of the m-th layer is,

$$2.1 \Rightarrow x_i^{(m)} = \sigma\left(u_i^{(m)}\right) = \sigma\left(\sum_j w_{ij}^{(m)} x_j^{(m-1)}\right). \tag{2.3}$$

The output of the neural network would thus be,

$$\hat{y}_j = x_i^{(k)} = \sigma\left(u_i^{(k)}\right).$$

It can be trivially proven that the negative gradient step would be,

$$\Delta w_{ij}^{(m)} = \eta \frac{\partial \ell(\mathbf{x}, \mathbf{y}; \mathbf{W})}{\partial \mathbf{W}} = \eta e_i^{(m)} x_j^{(m-1)}, \tag{2.4}$$

where e_i,

$$e_i^{(k)} = \frac{d}{d\mathbf{w}_i}\sigma\left(u_i^{(k)}\right)$$
$$e_i^{(m)} = \frac{d}{d\mathbf{w}_i}\sigma\left(u_i^{(m)}\right)\sum_j w_{ji}^{(m)} e_j^{(m+1)}, \quad m = 1, ..., k-1,$$

and η is the assigned learning rate.

2.1.5 Evaluation

The user of the ANN may want to sporadically test the accuracy of its predictions. So, after the pass of an arbitrary number of forward and backpropagation iterations it is deemed that an evaluation has to be performed. For this task the test dataset is used to determine whether the ANN is capable of accurate classification of previously unseen data.

Similarly to the loss function, during the evaluation, the predictions are compared to the actual labels of the data. But this time confidence is not a consideration. The label with the highest prediction probability is considered the network's "pick" and it is compared to the actual data item's label. The final accuracy value is the ratio of correctly labeled data, after the whole test dataset is processed by the network.

In machine learning research, it is common practice to use accuracy metrics to determine the termination of the learning process.

2.1.6 Summary

In summary, a neural network trains itself on pre-labeled data following these steps:

1. Initialization. The architecture of the ANN; the number of layers, perceptrons per layer, activation functions, loss function and hyper-parameters are chosen. The ANN variables (weights and biases) are randomly initialized.

2. Data processing. The whole dataset gets normalized and standardized. Data are split in training data and test data.

3. Forward propagation. A single or multiple data items are forward-passed through the neural network starting from the input layer. Eventually, the resulting output represents the network's prediction for labeling the given data item.

4. Loss calculation. The neural network's predictions from the previous step are compared to the desired predictions, which are determined by the actual labels of the data. This is accomplished using a loss function.

5. Backpropagation. The ANN estimates the gradient of the loss function layer by layer, commencing from the output layer and working backward. The opposite direction to the gradient determines the desired adjustments to the network's variables. The subsequent variable updates are performed.

6. Iteration. Repeat steps 3-5 until a stopping criterion is reached.

7. Evaluation. Validate the ANN to previously unseen data from the test dataset. Perform this step with subjective frequency.

2.2 Gradient Descent

The objective of Optimization is to identify the optimal element in the domain of an *objective function*, which results to this function's most favorable value, while obeying certain conditions. This ideal value is typically either the global minimum or the global maximum. Such problems are referred to as optimization problems and respectively either minimization or maximization problems. The following is the form of a generic minimization problem as termed by Boyd and Vandenberghe [6],

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_j(x) = 0, \quad j = 1, \ldots, p \quad , \end{aligned} \tag{2.5}$$

where f is the objective function, $g_i(x)$, for $i = 1, \ldots, m$ are the inequality conditions, $h_j(x)$, for $j = 1, \ldots, p$ are the equality conditions and $m, p \geq 0$. In the context of Machine Learning, we deal for the most part with minimization problems as we attempt to minimize a specific loss function.

Tracing back to 1847, the steepest descent method was first introduced by mathematician Augustin-Louis Cauchy as a means to compute the orbits of celestial bodies [21]. However, the computational form and the proof of convergence of *gradient descent* took shape later in the 20th century and variations of it were used to solve complex optimization problems. The core concept is straightforward: begin from a certain point within the domain of the function and gradually take steps towards the opposite direction of the function's gradient and thus towards a local minimum of the function. Of course, each subsequent step should be inside the domain of the function as well. This condition is ensured by calculating a step value using a line search algorithm[1]. The stopping criterion

[1]The backtracking line search is among the most straightforward and commonly used line search algorithms. Unlike the "vanilla" exact line search, it allows for the use of an arbitrary constant $\alpha \in \{0, 0.5\}$, which controls the size of updates and gradually decreases the step by a constant $\beta \in \{0, 1\}$ to abide by the Armijo-Goldstein condition. It won't be explained further, as it isn't applied to a ML setting. For more details refer to "Convex Optimization" by Boyd-Vandenberghe [6].

of the algorithm is reached, when the euclidean norm of the gradient reaches an arbitrary minimal value ϵ [6].

Algorithm 1 Gradient Descent

Given function f and starting point $\mathbf{x}_0 \in \mathbf{dom}\ f$
repeat
  $\Delta\mathbf{x} := -\nabla_{\mathbf{x}} f(\mathbf{x})$
  Line search to compute step $\mathbf{t}$
  $\mathbf{x} := \mathbf{x} + \mathbf{t}\Delta\mathbf{x}$
until $\|\nabla f(\mathbf{x})\|_2^2 \leq \epsilon$

It is important to note that Algorithm 1 guarantees convergence at the global minimum only in the case of strictly convex functions. A non-convex function implies the existence of local minima that can potentially trap the algorithm to a suboptimal solution. Additionally, classic gradient descent is not suitable for large amounts of data as it would require gradients' computation across the entire dataset for each update.

2.3 Gradient Descent Algorithms

Gradient descent is the cornerstone of most optimizers used in neural networks. The learning process is drawing upon a subset of samples $x^{(i)}$ and labels $y^{(i)}$ of a dataset $(\mathcal{D} = (\mathcal{X}, \mathcal{Y}))$ in every update step. The objective function in the context of ANNs is termed loss function, and it evaluates the discrepancy between the model's predictions and the actual labels. The loss function is differentiable and takes as its input the current neural network parameters, namely its weights w and biases b, and a subset of samples with their corresponding labels $(X \subseteq \mathcal{X}, Y \subseteq \mathcal{Y})$. So in general,

$$\ell : \mathbb{R}^k \to \mathbb{R},\ \nabla_\theta \ell(\theta; X, Y) \in \mathbb{R}^k$$
$$\theta = (w, b) \in \mathbb{R}^k,\ x^{(i)} \in \mathbb{R}^m,\ y^{(i)} \in \mathbb{R}^n.$$

The goal is to iteratively adjust the components of vector θ in order to achieve the optimal θ_*, which minimizes ℓ. This is the following minimization problem in the style of Equation 2.5,

$$\underset{\theta \in \mathbb{R}^k}{\text{minimize}} \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} \frac{1}{|X|} \ell(\theta; x, y).$$

Various optimizers achieve this through distinct update techniques. The size of each update step is computed based on Gradient Descent and it is also analogous to the subjective hyper-parameter of the learning rate (η). The learning rate has typically a value around $\{10^{-2}, 10^{-3}\}$. The optimizer terminates when a specific convergence criterion is reached.

2.3.1 Stochastic Gradient Descent

In *Stochastic Gradient Descent (SGD)* only one sample $x^{(i)}$ and its label $y^{(i)}$ is forward-passed through the ANN at every update step and so the sets X, Y are singleton. This means that the loss function will end up as a function of actual label $y^{(i)}$ and the predicted $\hat{y}^{(i)}$. For example, if the loss function is the L2 loss,

$$\ell(\theta; x^{(i)}, y^{(i)}) = \left(y^{(i)} - \hat{y}^{(i)}\right)^2.$$

Therefore, the update step of the model's parameters is,

$$\theta_t = \theta_{t-1} - \eta \cdot \nabla_\theta \ell(\theta_{t-1}; x^{(i)}, y^{(i)}). \tag{2.6}$$

For ease of reference, Equation 2.6 is presented as,

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \ell(\theta; x^{(i)}, y^{(i)})$$

and all following equivalent equations concerning update steps will do the same.

This algorithm is highly efficient in terms of updates, updating its parameters for each incoming sample, making it well-suited for online learning scenarios. The algorithm's shuffling of individual samples introduces variability into the training process, and this variability can enhance the model's generalization capabilities. Generalization refers to the model's ability to make accurate predictions on previously unseen test data. It can be proven that SGD is more likely to escape from a local minimum due to the randomness of sample selection [4]. On the other hand, SGD, with the limitation of not being able to leverage computational vectorization can result in lower training speed. Also, noise prevents the algorithm from converging.

2.3.2 Batch Gradient Descent

The algorithm analogous to classic mathematical gradient descent discussed in Section 2.1 is the naive *Batch Gradient Descent*. In this approach, the loss function is computed for the entire dataset in a single update ($X = \mathcal{X}$, $Y = \mathcal{Y}$).

$$\theta \leftarrow \theta - \eta \cdot \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} \frac{1}{|\mathcal{X}|} \nabla_\theta \ell(\theta; x, y)$$

On the positive side, Batch GD provides an unbiased estimation of the gradients, as it utilizes the entire dataset for each update. This ensures a trajectory that consistently moves towards a minimum, which is especially advantageous in scenarios with small datasets, such as IoT devices. However, the method comes with computational trade-offs. The computational cost of such large update steps can be prohibitive for large datasets. The gradient updates are time-intensive, making it less suited for online learning environments where rapid adaptation is critical. The accuracy of correctly predicting test data will remain low for the few first updates.

2.3.3 Mini-Batch Gradient Descent

In *Mini-Batch Gradient Descent* a *mini-batch* of samples is forward-passed through the ANN at each step [16]. The mini-batch of samples is a subset of samples of the dataset ($X \subset \mathcal{X}, Y \subset \mathcal{Y}$). This means that the update step should take into consideration the predictions of all samples in the mini-batch. The mini-batch size is generally a power of 2 in order to utilize GPU usage. So, the step loss is the average of all the losses in the mini-batch:

$$\ell(\theta; X, Y) = \sum_{x \in X, y \in Y} \frac{1}{|X|} \ell(\theta; x, y)$$

$$\theta \leftarrow \theta - \eta \cdot \sum_{x \in X, y \in Y} \frac{1}{|X|} \nabla_\theta \ell(\theta; x, y). \tag{2.7}$$

Mini-Batch Gradient Descent aims to combine the best of both SGD and Batch GD worlds. It generally outperforms Batch Gradient Descent in terms of speed while also providing good generalization by randomly shuffling the mini-batches. Forward-passing multiple samples allows vectorization and thus computational parallelism. Although it doesn't guarantee convergence like SGD, Mini-Batch Gradient Descent offers a similar frequency of online updates, contingent on the mini-batch size. Again this algorithm, similarly to SGD, cannot guarantee convergence, but it offers, depending on the size of the mini-batch, equivalently frequent online updates.

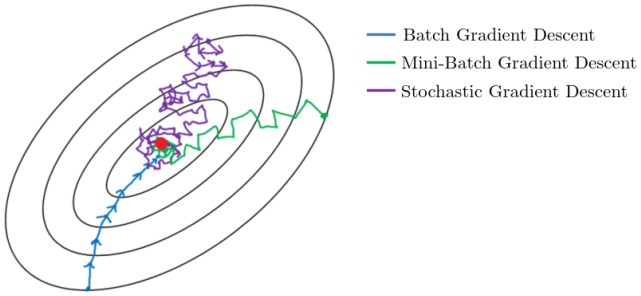


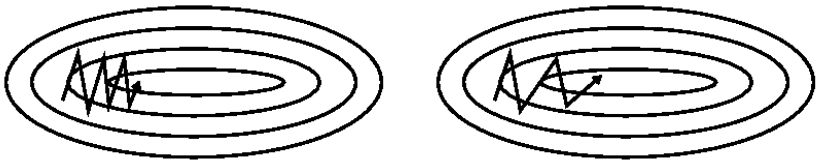
Figure 2.3: Trajectories of Gradient Descent Algorithms on Loss Function Contour
Source Ethan Irby

In Figure 2.3, the contour plot of the loss function is displayed. This function is convex and evidently has a global minimum at the center of the figure, indicated by a red dot. The three algorithms —Batch Gradient Descent, Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent— initiate from distinct starting points. Batch Gradient Descent consistently moves in the correct direction toward the minimum. In contrast, SGD follows a somewhat chaotic path, eventually nearing the minimum but oscillating around it. Mini-Batch Gradient Descent exhibits a trajectory similar to that of SGD, but distinguishes itself by taking larger and more precise steps toward the minimum.

2.4 Gradient Descent Optimizers

2.4.1 Momentum

SGD struggles with navigating *ravines*, i.e. regions where the surface exhibits a significantly steeper curvature along some dimensions compared to others. This can be seen in Figure 2.4a where the vertical dimension is much steeper and classic SGD gets carried away from the opposite direction of the function's gradient. Ravines are frequently observed around local minima and render classic SGD time-consuming especially for non-convex functions.



(a) Without Momentum (b) With Momentum

Figure 2.4: Trajectories with and without Momentum. Source Genevieve B. Orr

In order to mitigate this waving effect of the update steps, an *SGD Momentum* algorithm is being used. The result of the Momentum algorithm can be seen in Figure 2.4b. In its simpler form it makes use of a sequence v_t that calculates the exponentially weighed average of previous updates. For $\beta \in [0, 1]$ the update step is:

$$v_t = \beta v_{t-1} + \eta \nabla_\theta \ell(\theta_{t-1}, x^{(i)}, y^{(i)})$$
$$\theta \leftarrow \theta - v_t$$

2.4.2 Adam Optimizer

Adam (Adaptive Moment Estimation) is an optimizer method developed by Kingma and Lei Ba [20] and is elaborated in detail in Algorithm 2. Adam incorporates elements from both RMSprop and Adadelta [1] by maintaining an exponentially decaying average of past squared gradients, denoted as v_t. The vector v_t is referred to as 2^{nd} raw moment estimate and it accounts for the rate of gradients change, providing a dynamic learning rate specific to each parameter. Additionally, it retains an exponentially decaying average of past gradients, m_t, akin to the concept of the Momentum method. The variable m_t is called 1^{st} moment estimate and smooths the optimization path when gradient changes directions. Both of these two vectors undergo bias-correction at each timestep based on the current timestep t.

Adam has gained widespread popularity as an optimizer in the realm of Machine Learning due to its unique blend of advantages borrowed from SGD, Momentum, RMSprop, and Adadelta. Its adaptivity is well-suited for tackling non-convex optimization

[1] RMSprop [36] and Adadelta [40] were independently created around the same period. Both algorithms employ gradient normalization through the root mean square of previous gradients, allowing them to dynamically adjust the learning rate during the training process.

challenges and handling problems with sparse gradients, which are not uncommon in the landscape of ML problems. Furthermore, the adaptive learning rate component in Adam streamlines the training process by reducing the need for extensive hyper-parameter tuning, ultimately saving valuable time and resources in model development.

Algorithm 2 Adam (Adaptive Moment Estimation) [20]

Hyper-parameters

η: learning rate

$\beta_1, \beta_2 \in [0, 1)$: exponential decay rates for first/second moment estimates

Initially

$t = 0$ ▷ Timestep

$m_0 = 0$ ▷ 1^{st} moment vector

$v_0 = 0$ ▷ 2^{nd} moment vector

repeat

$g_t = \nabla_\theta \ell(\theta_{t-1}, x, y)$ ▷ Get gradients w.r.t. stochastic objective at timestep t

$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Update biased 1^{st} moment estimate

$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t \odot g_t$ ▷ Update biased 2^{nd} raw moment estimate

$\hat{m}_t = m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected 1^{st} moment estimate

$\hat{v}_t = v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected 2^{nd} raw moment estimate

$\theta_t = \theta_{t-1} - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ ▷ Update parameters

$t = t + 1$

until θ_t converges

2.5 Distributed Machine Learning

Traditional non-distributed ML is performed centrally on a single machine. Although the raw data might originate from various locations, it must be collected and transmitted over the network to a central processing unit. In this setup, a single neural network handles the entire learning process, as shown in Figure 2.5a. While this initial technique may have a simple implementation and preprocesses data that can be easily generalized, it comes at a cost to performance, fault tolerance, scalability and privacy [13].

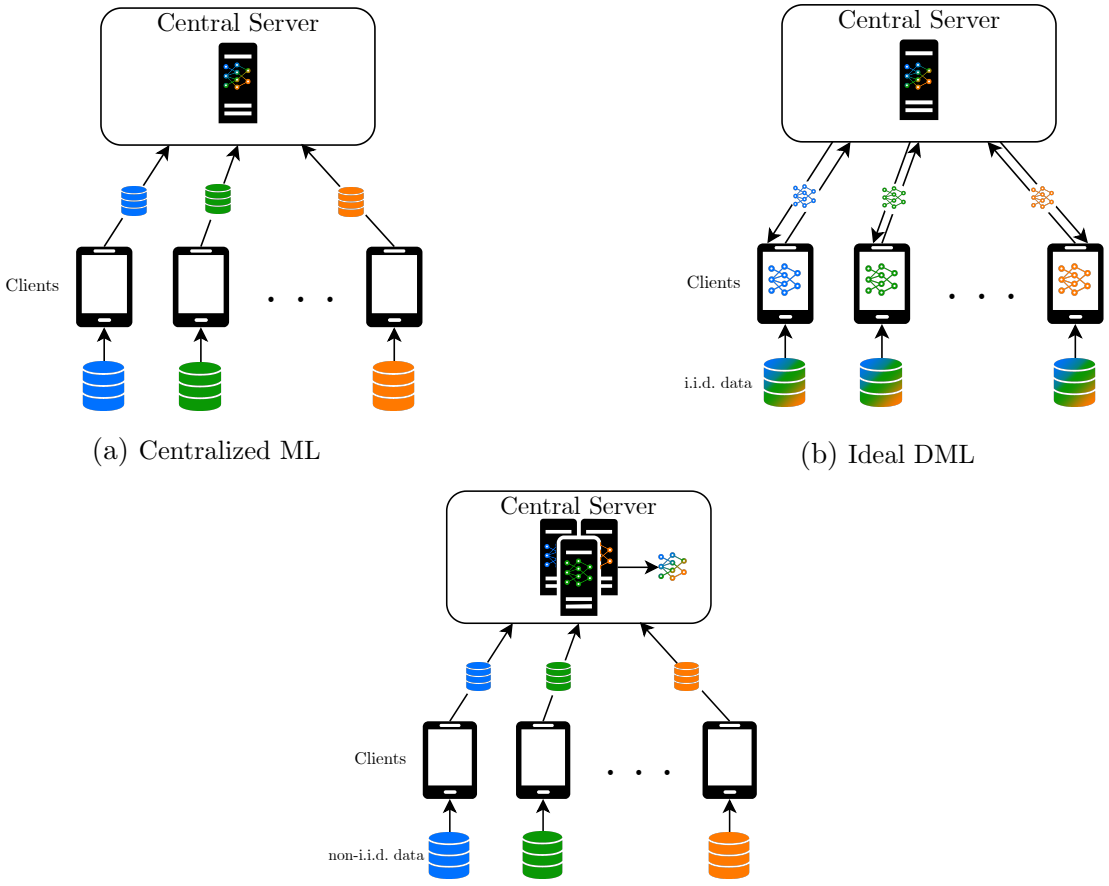
Contrarily, *Distributed Machine Learning (DML)* is executed in multi-node networks designed to improve performance, increase accuracy, and scale to larger input data sizes [14]. Each client is responsible for training its own local neural network based on its local data. At regular intervals, these models must pause their independent learning and engage in synchronization, wherein they exchange their neural network variables with one another. This synchronization process takes place at the conclusion of a communication round. Synchronizing with peers can happen either via peer-to-peer model averaging [37, 34] or by using a centralized parameter server [23]. The outcome of this averaging process is a global neural network that encapsulates the collective learning progress achieved by all nodes.

This indicates that DML can take advantage of data and model parallelization, providing faster training, scalability, and minimized network overhead. Learning on models is done over mini-batches, which are subsets of training data for each node and iteration.

There is no need to communicate a large quantity of potentially vulnerable raw data through the network to reach a global model, as this is achieved by averaging the local models of each node.

The primary limitation of classic DML algorithms, employing straightforward aggregation techniques, is their dependency on the dataset being *independent and identically distributed (IID)* across all clients. When dealing with IID data, learning can occur effectively on the edge devices or clients, as illustrated in Figure 2.5b. Only the neural network variables are transmitted across the network in this scenario.

However, in many instances, these basic DML algorithms perform better in a cluster-based environment, as depicted in Figure 2.5c. In this setup, whether the data is IID or not, it is collected centrally on a server, where it is merged into a single dataset and distributed to multiple clients within a cluster of computational nodes. This can be the only option because IID data are not usually the case when training occurs in many individual clients in poor network conditions and with a variety of local data. While the computational aspect of the process remains distributed, data collection and learning become centralized. This approach raises concerns about client privacy since raw data still need to traverse the network. Those problems get resolved using Federated Learning (FL) algorithms, which will be explained in detail in Section 2.6.



(a) Centralized ML

(b) Ideal DML

(c) Cluster-based DML

Figure 2.5: Distributed Machine Learning Networks

2.5.1 Distributed Machine Learning Algorithms

A multitude of algorithms have been developed for Distributed Machine Learning. These algorithms are designed so that they parallelize the learning in an efficient manner, are fault tolerant, preserve data privacy and reduce communication overhead. They have to simultaneously provide high training accuracy and low network overhead/traffic.

Large network bandwidth consumption needs to be diminished as data centers and devices have limited networking capabilities and receive a high frequency of data, especially due to GPU technology advancements, e.g., CUDA cores. At the same time, DML adds complexity and expensive synchronization barriers to ML, which need careful consideration. In any case, stragglers, single points of failure, and competing tenants should be avoided during the development of algorithms. In general, the steps a DML algorithm takes are the following:

1. Initialization. At first all local models are initialized typically with random weights and biases.

2. Local Training. The clients start training their models using their local dataset as long as the current communication round lasts.

3. Synchronization & Aggregation. All clients synchronize at the end of the communication round by broadcasting their model variables to their peers or to a central server [2]. Then, the variables get aggregated, mainly by averaging, to create a global model.

4. Global Model Update. The global model is transmitted through the network and all clients' models get updated to the global model.

5. Iteration. Iterate through steps 2-4 until a stopping criterion is reached.

2.5.2 Asynchronous Stochastic Gradient Descent

The simplest form of DML algorithms is the *Asynchronous Stochastic Gradient Descent (ASGD)* algorithm. A lot of state of the art algorithms are using the basics of ASGD with some variation. Some of the most popular are Hogwild! [29], which was the pioneer in the field, Downpour SGD [11], that introduces a parameter server with multiple model replicas and the asynchronous version of Elastic Averaging SGD [41], which dynamically decides the duration of local training for each client, reducing network overhead. We won't get into detail explaining the nuances of these algorithms and instead we will focus on the generic form of an asynchronous algorithm for a DML system.

As the name suggests the clients don't have to synchronize in order to start local training. The server updates the global model every time it receives variables from any client and it isn't confined, waiting for all clients to communicate with it. In this case, communication rounds are not essential to the algorithm and are not clearly defined.

[2]Some algorithms train collectively with the absence of a central server. This applies to the KungFu library that we have used in our experiments and is examined in more detail in Chapter 5.

For this example training happens with a simple Mini-Batch Gradient Descent optimizer, which is the most popular choice for such an algorithm. Mini-Batch GD has been further explained in Section 2.3.3 and Equation 2.7 describes its update step. It was mentioned earlier, that when clients communicate with the central server, they transmit their model variables, i.e., weights and biases. In Chapter 5, we will primarily employ this technique of weight synchronization, as this is the methodology followed in our experiments. However, it's important to note that an alternative approach exists, which involves transmitting the gradient of the model's loss function. This approach is equivalent to the previous and its popularity within the literature is the main reason for its use in this context. So, Equation 2.7 gets replaced by,

$$G \leftarrow G + \sum_{x \in X, y \in Y} \frac{1}{|X|} \nabla_{\mathbf{w}} \ell(w; x, y),$$

where X is the mini-batch and $|X|$ the size of it, G is the stochastic local gradient and (x, y) is a data point of the mini-batch with its respective label. In Chapter 5, we will use the weights synchronization technique that we used on our experiments. Algorithm 3 describes ASGD on the client side and Algorithm 4 ASGD on the server side [3].

Algorithm 3 Asynchronous SGD Client k

Initially $\mathbf{w}_k$ = random values
repeat
  Read $\mathbf{w}_{global}$ variables from server
  $\mathbf{w}_k \leftarrow \mathbf{w}_{global}$
  for $i = 1, \ldots, |X|$ **do**
    $G_k \leftarrow G_k + \frac{1}{|X|} \nabla_{\mathbf{w}} \ell(\mathbf{w}_k; x^{(i)}, y^{(i)})$
  end for
  Send G_k gradient to server
until stopping criterion is reached

Algorithm 4 Asynchronous SGD Server

Initially $\mathbf{w}_{global}$ = random values
repeat
  Wait for G_k gradient of any client k
  $\mathbf{w}_{global} \leftarrow \mathbf{w}_{global} - \eta \cdot G_k$
until stopping criterion is reached

Asynchronous SGD has the ability to swiftly update the global model and, subsequently, speed up the local learning process. This is because asynchronous updates happen instantly without waiting for other client updates. This attribute can prove to be advantageous in scenarios where *straggler* clients are prevalent. Straggler clients are clients that delay their communication with the server, due to either slow-paced local training or network limitations. In a synchronous environment these stragglers would have significantly slowed down the collective learning process. Another benefit in using an asynchronous algorithm is the ability to effortlessly scale the network of clients. The number of clients in the network is irrelevant to the aggregation process.

The major drawback of Asynchronous SGD is the concept of staleness. Local training on a client starts by reading the model variables from the server. Nevertheless, over the course of local training, the global variables may undergo substantial alterations,

[3] In these examples synchronization happens for each batch step, but in general this might not be the case. Some variants of Asynchronous SGD allow clients to explore locally for more than one step before synchronization.

potentially rendering the client's gradient updates outdated to the point of impacting the effectiveness of the algorithm.

2.5.3 Synchronous Stochastic Gradient Descent

The alternative to asynchronous algorithms are *synchronous* algorithms for DML. Noteworthy examples of synchronous algorithms include Parallel SGD [19], which is the vanilla synchronous SGD, the synchronous version Elastic Averaging SGD [41], which was already mentioned in Section 2.5.2, and Large Minibatch SGD [16], that found a linear scaling rule between mini-batch size and learning rate.

In this section, our primary focus is again a generic version of *Synchronous Stochastic Gradient Descent (SSGD)* algorithm. Similar to the asynchronous algorithm outlined in Section 2.5.2, the gradient of the local model is the one that traverses the network. The key distinction of this algorithm lies in the fact that during each communication round, the server pauses to accumulate the local updates from all clients prior to proceeding with aggregation and the subsequent global update. The clients that have already transmitted their local updates similarly pause, awaiting the completion of the round. Algorithm 5 describes SSGD on the client side and Algorithm 6 SSGD on the server side [3]. Note that on the server side the global model update step happens only when n gradients are received, n being the number of clients in the network.

Algorithm 5 Synchronous SGD Client k

Initially $\mathbf{w}_k$ = random values
repeat
  Wait for $\mathbf{w}_{global}$ variables
  $\mathbf{w}_k \leftarrow \mathbf{w}_{global}$
  for $i = 1, \ldots, |X|$ **do**
    $G_k \leftarrow G_k + \frac{1}{|X|} \nabla_{\mathbf{w}} \ell(\mathbf{w}_k; x^{(i)}, y^{(i)})$
  end for
  Send G_k gradient to server
until stopping criterion is reached

Algorithm 6 Synchronous SGD Server

Initially $\mathbf{w}_{global}$ = random values
repeat
  $\mathcal{G} = \{\}$
  while $|\mathcal{G}| < n$ **do**
    Wait for G_k gradient of any client k
    $\mathcal{G} \leftarrow \mathcal{G} \cup \{G_k\}$
  end while
  $\mathbf{w}_{global} \leftarrow \mathbf{w}_{global} - \eta \cdot \frac{1}{n} \sum_{k=1}^{n} G_k$
  Send $\mathbf{w}_{global}$ variables to all clients
until stopping criterion is reached

It is obvious that this implementation offers more consistency as clients update their local variables in unison and train based on an identical version of the global model. Data being up to date eliminate any potential detriments to the collective learning process. Furthermore, it is much simpler to debug and devise smart dynamic strategies on this algorithm. Given that all clients maintain a uniform state at any given moment, synchronization for particular tasks becomes straightforward. For these reasons, we choose to implement the Functional Dynamic Averaging methods, explained in Section 4.2, using a synchronous algorithm, more on that in Chapter 5.

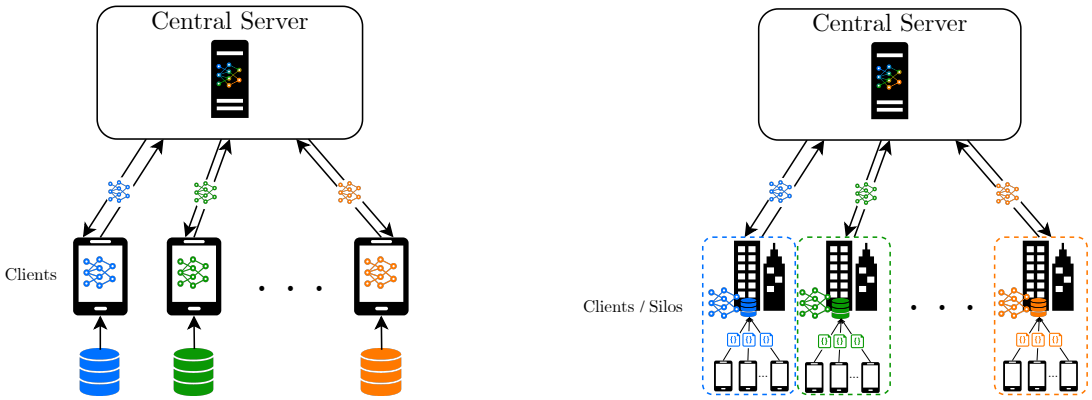
The main shortcoming of such synchronous algorithms lies in the presence of straggler clients, which can substantially impede the synchronization process. The completion of each communication round is dependent upon the receipt of updates from all participating

clients. This is why some variations of the synchronous algorithm try to avoid potential straggler issues by waiting to receive gradients for a subset of all clients [10]. The slower workers' updates are dropped when they arrive and thus latency is limited.

2.6 Federated Learning

Federated Learning (FL) is a form of Distributed Machine Learning, with an additional emphasis on data privacy [22]. Local data are strictly contained in the clients and are not transmitted through the network. Only the model variables are send to central servers. This can be profoundly beneficial for personal mobile devices since learning can be achieved using cross-user data, while simultaneously abiding by personal data security laws and regulations.

Data is maintained locally, either on edge devices (Figure 2.6a), e.g., mobile phones, sensors, medical devices, or in so-called data silos (Figure 2.6b), e.g., different departments of a company having separate independent databases or a network of separate hospitals. Data silos can be proven effective in areas where data security plays a significant role, and sensitive raw data should be confined in certain physical locations. Such domains encompass sectors like healthcare, smart cities facilitated by IoT applications, and military establishments. Generally, the goal of Federated Learning is to adapt distributed machine learning for a vast network of diverse devices, often with limited computational power and each processing relatively small datasets.



(a) Cross-Device FL (b) Cross-Silo FL

Figure 2.6: Federated Learning Architectures

Federated Learning algorithms resemble DML algorithms, but prioritize edge computing. The learning process should predominantly take place on edge devices. As a result, it is crucial for such algorithms to incorporate methodologies that address the following key challenges:

1. *Enable learning from data that is neither independent nor identically distributed.*

   In most cases, in Federated Learning applications, it's typical to have vast numbers of clients participating, with each client often corresponding to an individual user.

The user base can encompass millions spanning multiple diverse countries, cultural backgrounds and professions. Hence, the data generated and utilized for training on each client tends to have significant variance. For example, take the Gboard application by Google that employs FL to enhance virtual keyboard recommendations. The vocabulary used by young users differs greatly compared to older ones, language can differ around the globe and even from region to region. Consequently, the data in the network are neither independent nor identically distributed, posing challenges to the efficient convergence of the training process. A variety of distinct techniques have been devised, each aiming to overcome a certain category of data skewness. A detailed overview of these methods is presented in "Federated learning on non-IID data: A survey" by Hangyu Zhu et al. [42].

2. *Reduce the communication overhead associated with synchronization.*

   The sheer magnitude of participating clients, many operating on less-than-ideal wireless network infrastructure, necessitates significant reduction of network communication traffic. Merely transmitting model variables through the network doesn't sufficiently address the demands of Federated Learning. Therefore, it should be guaranteed that global updates aren't mandated at every training iteration. To the contrary, local models should be allowed the flexibility to independently explore more of their own data, guided by global metrics. If these global metrics exceed an arbitrary threshold, synchronization is initiated and a new communication round follows. In this thesis, we consider the divergence between the local models and their average as a global metric.

3. *Minimize the risk of malicious identification of individual data points.*

   On initial consideration, it may seem far-fetched to have concerns about privacy, especially given that no raw data travel through the network. Despite the absence of direct data transmission, malicious actors could potentially gain access to model data stored on a central server and deduce the input dataset. This technique is termed a "model inversion attack", where an adversary designs a neural network inverse to the global one. This adversarial network takes the global model's output as its input and aims to reconstruct the initial data points of individual users. While the efficacy of such attacks tends to increase with fewer clients, adversaries can still have insights into the collective dataset even in large networks. Knowing the peculiarities of the dataset someone can design adversarial clients that falsely label data (data poisoning). FL systems are often equipped with further privacy mechanisms to avoid such attacks. Most notorious is the work done in the Differential Privacy front. A differentially private algorithm ensures that by gaining access to a global model's output it is not possible to determine whether an individual user has contributed or not in the learning process. This is performed by adding extra noise to model updates. We won't get into detail on differentially private algorithms as it is out of this thesis scope. If you want to have an in-depth understanding on the dangers looming over FL and DML, you are suggested to read the article titled "Vulnerabilities in Federated Learning" by N. Bouacida and P. Mohapatra [5].

2.6.1 Federated Learning Algorithms

In order to clearly understand the inner workings of Federated Learning algorithms we ought to describe the general strategy that all such algorithms use, and also describe some of the more basic ones in detail. Every Federated Learning Algorithm follows the steps detailed below and in this order.

1. Initialization. Initialize the global model and local models with random variable values.

2. Client Selection. Choose a subset of clients to participate in the current communication round. This selection process may be random or deliberately based on specific criteria, such as the device's availability or the quality of its network connection.

3. Broadcast. The chosen clients retrieve the latest global model from the central server.

4. Local Training. Each selected client computes the next local training step based on its local dataset.

5. Synchronization & Aggregation. All participating clients synchronize at the end of the communication round by broadcasting their model variables to the central server. Then, the variables get aggregated, mainly by averaging, to update the global model.

6. Global Model Update. The global model gets updated with the new aggregated values of the chosen clients.

7. Iteration. Iterate through steps 2-5 until a stopping criterion is reached.

The conventional practice for client selection in a communication round within Federated Learning involves defining a fraction of the total set of clients, represented by a variable C within the range of $(0, 1]$. Consequently, only a C-fraction of the total client population —expressed as $C \cdot n$, where n is the total number of clients— is designated to train during each round.

2.6.2 Federated Stochastic Gradient Descent

Federated Stochastic Gradient Descent (FedSGD) [27] is the simplest Federated Learning algorithm. It is used as a baseline in the field's literature in order to compare with more complicated algorithms. This algorithm is almost identical to the simple Synchronous SGD algorithm (Section 2.5.3), as it uses plain Stochastic Gradient Descent for training and has a synchronous global update mechanism. The fraction of participating clients is a constant $C = 1$, and so, all clients are participating at every round.

The main difference is that this algorithm uses the whole local dataset $\{\mathcal{X}_k, \mathcal{Y}_k\}$ of each client k to take a training step. So, at each communication round all clients calculate

their gradients G_k over their whole dataset and send it to the server. Then, the server calculates its weight variables $\mathbf{w}_{global}$ by taking an opposite step towards a weighted sum of the gradients. Each gradient is weighted by $\frac{d_k}{d}$, where $d_k = |\mathcal{X}_k|$ is the number of data points in client k dataset and n is the number of data points of all clients in the network. The pseudocode for FedSGD is given below in Algorithm 7 for the client-side and in Algorithm 8 for the server-side.

Algorithm 7 Federated SGD Client k

Initially $\mathbf{w}_k$ = random values
repeat
  Wait for $\mathbf{w}_{global}$ variables
  $\mathbf{w}_k \leftarrow \mathbf{w}_{global}$
  for $\{x, y\} \in \{\mathcal{X}_k, \mathcal{Y}_k\}$ **do**
    $G_k \leftarrow G_k + \frac{1}{d_k} \nabla_{\mathbf{w}} \ell(\mathbf{w}_k; x, y)$
  end for
  Send G_k gradient to server
until stopping criterion is reached

Algorithm 8 Federated SGD Server

Initially $\mathbf{w}_{global}$ = random values
repeat
  $\mathcal{G} = \{\}$
  while $|\mathcal{G}| < n$ **do**
    Wait for G_k gradient of any client k
    $\mathcal{G} \leftarrow \mathcal{G} \cup \{G_k\}$
  end while
  $d \leftarrow \sum_{k=1}^{n} d_k$
  $\mathbf{w}_{global} \leftarrow \mathbf{w}_{global} - \eta \sum_{k=1}^{n} \frac{d_k}{d} G_k$
  Send $\mathbf{w}_{global}$ variables to all clients
until stopping criterion is reached

2.6.3 Federated Averaging

Federated Averaging (FedAvg) [27] is one of the first ever Federated Learning algorithms that sought to increase robustness to non-IID data distributions. To further explain this algorithm, we have to introduce constants E, the number of epochs of local training performed at each communication round, and B, the local mini-batch size of each client. When using the term epoch in Machine Learning we are referring to a complete pass through the entire training dataset while training. So for FedSGD, in Section 2.6.2, we can say that $E = 1$ and $B \to \infty$.

For FedAvg, $C \neq 1$, and so at every communication round $m = C \cdot n$ clients participate in training. The server randomly selects m clients in a set S_t to train for the current round t. It then sends the global weight variables $\mathbf{w}_{global}$ to the selected clients in set S_t and commands them to initiate their local training. Each client splits its local dataset $\{\mathcal{X}_k, \mathcal{Y}_k\}$ to B-sized batches. Local training lasts for E complete passes of the local training dataset. At the end of each batch training the local weights are updated as we previously saw in Mini-Batch SGD and equation 2.7,

$$\mathbf{w}_k \leftarrow \mathbf{w}_k - \eta \cdot \sum_{x \in X, y \in Y} \frac{1}{B} \nabla_{\mathbf{w}} \ell(\mathbf{w}_k; x, y), \tag{2.8}$$

where $\{X, Y\}$ is a batch taken from $\{\mathcal{X}_k, \mathcal{Y}_k\}$. Then, each participating client proceeds to send the local updated weights to the central server. There, and when all clients in set S_t have returned their weights, the global model's weights are updated based on the transmitted local weights. Similarly to FedSGD, the global weights $\mathbf{w}_{global}$ are calculated

as a weighted sum of the participating clients weights. The coefficients are determined again by $\frac{d_k}{d}$, where d_k is the number of data points that client k used in training and d the number of data points that all participating clients in S_t used. The detailed algorithms for FedAvg are given below in Algorithm 9 for the client-side and in Algorithm 10 for the server-side.

FedAvg outperforms FedSGD. With much fewer communication rounds and by training subsets of clients at a time, FedAvg achieves training accuracy equivalent to the one achieved by FedSGD. Given the appropriate parameter tweaking communication rounds can be reduced almost 100 times. We aim at achieving similar results in our own experiments, where we seek to determine each communication round's duration dynamically.

Algorithm 9 Federated Averaging Client k

Initially $\mathbf{w}_k$ = random values
repeat
  Wait to be selected and get $\mathbf{w}_{global}$ variables from server
  $\mathbf{w}_k \leftarrow \mathbf{w}_{global}$
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size $|X|$)
  for each local epoch $e = 1, \ldots, E$ **do**
    for batch $b \in \mathcal{B}$ **do**
      for $i = 1, \ldots, |X|$ **do**
        $G_k \leftarrow G_k + \frac{1}{|X|} \nabla_{\mathbf{w}} \ell(\mathbf{w}_k; x^{(i)}, y^{(i)})$
      end for
      $\mathbf{w}_k \leftarrow \mathbf{w}_k - \eta\, G_k$
    end for
  end for
  Send $\mathbf{w}_k$ variables to server
until stopping criterion is reached

Algorithm 10 Federated Averaging Server

Initially $\mathbf{w}_{global}$ = random values
repeat
  $m \leftarrow \max(C \cdot K, 1)$
  $S_t \leftarrow$ (random set of m clients)
  Send $\mathbf{w}_{global}$ variables to m clients $\in S_t$ and initiate local training
  $\mathcal{W} = \{\}$
  while $|\mathcal{W}| < m$ **do**
    Wait for $\mathbf{w}_k$ variables of any client $k \in S_t$
    $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathbf{w}_k\}$
  end while
  $d \leftarrow \sum_{k \in S_t} d_k$
  $\mathbf{w}_{global} \leftarrow \sum_{k \in S_t} \frac{d_k}{d} \mathbf{w}_k$
until stopping criterion is reached

Chapter 3

Selected Tools and Framework

At the outset of this diploma thesis, it was quite clear that we needed the appropriate tools to implement a genuine federated learning system. Consequently, it was necessary to identify tools with inherent FL capabilities that could be easily scaled up into a fully functional deployment. Fortunately, there was an abundance of distributed machine learning and federated learning tools at our disposal, and the need for development from scratch was obviated. The pivotal decision was selecting the most suitable tool from the available options.

Starting from the bare bones, it was quite evident that Python would be the programming language of choice, as it is convenient for the use of data structures and provides, out of the box, libraries ideal for big data and machine learning operations.

In the context of machine learning frameworks, the deliberation was primarily between TensorFlow [25] and PyTorch [30]. Despite PyTorch's ease of use, Python-like code functions and simplicity, which make it favorable for use as a theoretical research tool, TensorFlow seemed more capable of real-life applications. TensorFlow has a much wider use in the Machine Learning industry as it is production-ready, especially for mobile devices, and it has comprehensive documentation and support. Industry leaders like Google, X (formerly known as Twitter) and Airbnb, among others, use TensorFlow for their Machine Learning and Artificial Intelligence applications.

Then, the decision had to be made on which framework to use in order to transition machine learning processes into a distributed and subsequently federated environment. The framework that we've concluded on is KungFu [7, 24] developed by Luo Mai et al., from Imperial College London's LSDS Group. It is a distributed machine learning library for TensorFlow, that enables adaptive training and provides distributed systems operations. We've opted in favor of KungFu, because it can be easily deployed right away on any diverse group of computing nodes, regardless of hardware specifications, physical and networkwide location. It promises high throughput speeds and equips the user with the ability to arbitrarily set network topologies. Arranging clients in hierarchical topologies could prove useful in implementing Federated Learning scenarios.

3.1 TensorFlow

TensorFlow is an interface for machine learning algorithms expression. It focuses on providing comprehensive machine learning solutions on a broad variety of platforms, from computationally weak mobile devices to large-scale distributed systems. It is designed to scale the training process across different types of hardware as CPUs, GPUs, FPGAs and TPUs, Google's custom Tensor Processing Units.

The TensorFlow development team has aimed to simplify its usability for real-world machine learning applications by crafting a unified system that serves both research and production needs. By the use of the Keras [8] open-source library, it equips the user with standard datasets, data pipelines and tools to easily validate and transform large datasets. TensorBoard, TensorFlow's visualization toolkit tracks training in real-time and offers immediate insight of the process. TensorFlow Distribute is an API to distribute training across multiple devices and includes multiple strategies, "flavors" of distributed machine learning.

TensorFlow is an open-source platform, released under the Apache 2.0 license, providing users with access to its API and reference implementation. It boasts a substantial user community, which actively fosters contributions from its members. The development on the interface is continuous since its release and its documentation is rich in examples and always up-to-date (TensorFlow API docs).

The touchstone of TensorFlow operations, as its name would suggest are tensors. Tensors are typed, multidimensional arrays, with types including signed and unsigned integers with sizes varying from 8 bits to 64 bits, float, double and strings.

TensorFlow —especially on its earlier 1.x versions— offers high computational performance and portability outside Python using a technique called graph execution. A TensorFlow graph is a directed graph that encapsulates a computation, comprising a series of nodes, with each node depicting a distinct mathematical operation. Upon the start of execution, clients generate this computational graph to serve as a pipe for the dataflow, with tensors being the entities that move along the graph's edges. The graph's structure is static during execution, which allows TensorFlow to possess a complete overview of a tensor's lifecycle, enabling it to make informed decisions regarding GPU usage and memory allocation. Parallelization of graphs can happen by distributing operational nodes to subgraphs and replacing cross-thread and cross-device edges with *Send* and *Receive* nodes. Only parts of computations that are independent can be split into different threads and devices.

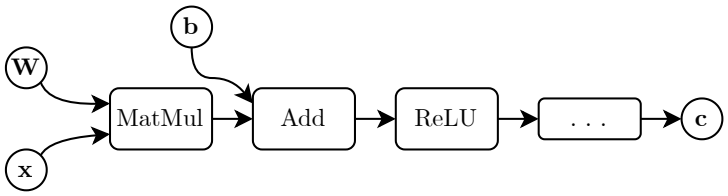


Figure 3.1: Tensorflow Computation Graph Example
Source [25]

With the advent of TensorFlow versions 2.x, the framework has introduced an alter-

native to graph execution known as eager execution. This approach allows TensorFlow operations to be performed sequentially by Python, providing immediate results in a build-as-you-go fashion. This interactive approach of eager execution can be particularly beneficial for beginners or during the prototyping phase of projects, offering a more intuitive and accessible way to work with TensorFlow.

3.2 Keras

Keras [8] is the high level API of the TensorFlow framework. Keras is designed to be user-friendly without it being over-simplistic. It helps developers by standardizing the trivial parts of training, allowing them to concentrate on the critical aspects of their projects. It embodies the principle of progressive disclosure of complexity; basic tasks are straightforward and easy to express, while increase in difficulty is gradual. Moreover, Keras delivers robust performance and scalability. This is evident by its wide adoption in industry and research.

The foundational data structures of Keras are *layers* and *models*. Models are comprised of layers and are the way, to design and initialize neural networks on TensorFlow. The Sequential model stands as the most basic form of model provided by Keras, essentially being a linear stack of layers. The *model* structure grants essential and handy functions for training the model (*fit*) and accuracy evaluation (*evaluate*). Keras, also, offers straightaway popular optimizers, loss functions and metrics for their models.

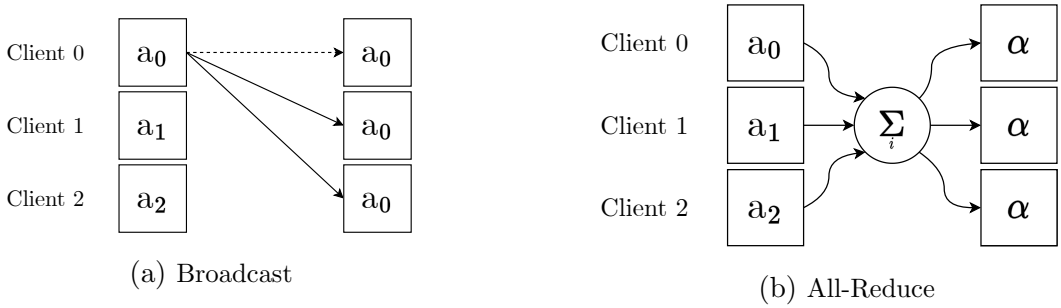
3.3 KungFu

KungFu [7, 24] is a distributed machine learning library for TensorFlow. Its main focus is to provide a distributed machine learning tool that can perform fast inter-GPU communication, while also enabling its users to dynamically tune hyper-parameters online during the training process. Machine Learning users lose a significant amount of their labor time configuring training hyper-parameters. There are no clean-cut solutions for parameter tuning, as each ML problem has its own unique dataset and requirements. Users necessarily resort to exhaustive trial and error techniques. KungFu comes to fill this gap in existing distributed ML tools.

KungFu adapts training parameters by expressing certain *Adaptation Policies (APs)*. In order to convey these APs KungFu is equipped with distributed functions or operations. These functions can be divided into three categories; monitor, communication and adaptation functions. Communication functions allow for tensor transmission through the network and adaptation functions can dynamically alter hyper-parameters' values. Monitor functions can be developed for each user's custom use case and offer global metrics that can initiate parameter adaptations.

Some of the most useful operations provided for our training project are *broadcast()*, *allreduce()*, *rank()* and *size()*. Functions *rank()* and *size()* are adaptation functions that output the "identification number" of each client in the network and the total number of clients in the network at the given moment respectively. Functions *broadcast()* and

allreduce() are communication functions. The *broadcast()* function is self-explanatory; a client broadcasts a variable and after the operation all other clients are assigned its value. On the other hand, the *allreduce()* function receives input from all clients, aggregates them using a function and at the end distributes its output to all clients. The KungFu *allreduce()* operation outputs the sum of all clients' values.



(a) Broadcast (b) All-Reduce

Figure 3.2: Communication functions

Collective operations in KungFu are performed without a central server or coordinator. Clients communicate with each other in a decentralized manner by peer-to-peer communication. This is accomplished by an *asynchronous collective communication layer* implemented in the Go and C++ programming languages, which is then bound with Python to run experiments using TensorFlow or PyTorch. Decentralized communication is also accelerated —when communicating gradients— by the parallel use of the Nvidia Collective Communications Library (NCCL) alongside the communication layer. NCCL is following on the Message Passing Interface (MPI) [28] and achieves peer-to-peer communication by designing a bidirectional ring topology based on the actual topology of the network's GPUs [38]. GPU cores and nodes that neighbor each other are more likely to be paired together in the ring [17]. KungFu's communication network is proven to perform better than the state-of-the-art Horovod [34] in terms of throughput.

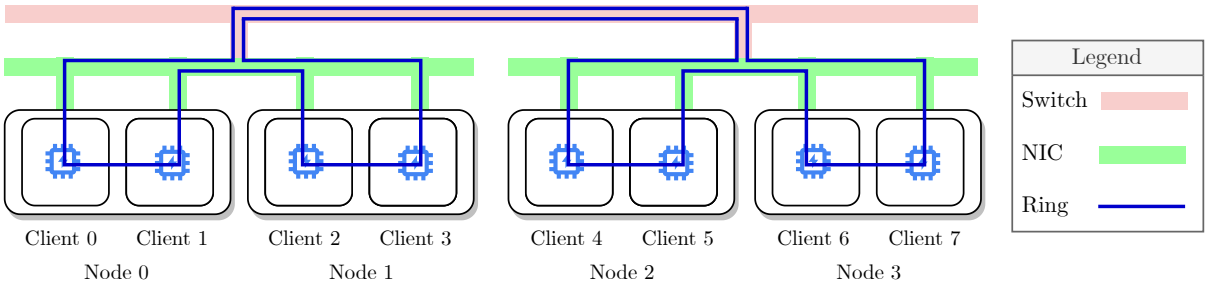


Figure 3.3: Ring Topology Inspired by [17]

Apart from the automatic ring topology detection, KungFu offers the adaptation function *tree()*, to set custom tree topologies. This function receives as input a Python list that contains the topology information in the form $[\text{parent}_0, \text{parent}_1, \ldots, \text{parent}_n]$. The i-th element of the list contains the i-th client's parent in the topology, while the client that is its own parent is a tree root. Another adaptation function is *set_size()*, which can decrease and increase the number of clients in the network.

Chapter 4

Methodology

4.1 Geometric Approach

Izchak Sharfman et al. have proposed and validated an innovative strategy for overseeing threshold functions within distributed systems in their work "A *Geometric Approach* to Monitoring Threshold Functions over Distributed Data Streams" [35]. The main idea behind it is that an arbitrary global monitoring task can be split into a set of constraints. These constraints are implemented on individual data streams to locally eliminate data increments that are irrelevant to the monitoring result. Consequently, communication between different nodes gets reduced significantly while the global monitoring quality remains high.

We will swiftly clarify the decentralized aspect of the Geometric Approach (GA) for better understanding. Keep in mind that we will refer to data, either local or global as *statistics* and to clients as *nodes*, as this is the terminology used for the GA. Let there be a network of n nodes and let the *local statistics* of each node p_k at moment t be $\mathbf{v}_t^{(k)} \in \mathbb{R}^d$ for $k \in [1, n]$. The *global statistics* vector would be trivially calculated as,

$$\overline{\mathbf{v}}_t = \frac{\sum_{k=1}^{n} w^{(k)} \mathbf{v}_t^{(k)}}{\sum_{k=1}^{n} w^{(k)}},$$

where $w^{(k)}$ is the arbitrary weight of node's p_k statistics. To monitor the global statistics vector would require the continuous communication of the local statistics at each t round. This is why each node stores locally the last statistics vector as $\mathbf{v}'^{(k)}$ from all nodes, as well as a local estimate $\mathbf{e}_t$ of the global statistics. The estimate $\mathbf{e}_t$ is calculated as,

$$\mathbf{e}_t = \frac{\sum_{k=1}^{n} w^{(k)} \mathbf{v}'^{(k)}}{\sum_{k=1}^{n} w^{(k)}} \quad .$$

In addition to these variables, each node stores a statistics *delta vector* $\Delta\mathbf{v}_t^{(k)} = \mathbf{v}_t^{(k)} - \mathbf{v}'^{(k)}$, which is the difference between the current local statistics vector and the last

statistics vector this node broadcasted to the network. Another parameter stored locally is the so-called *drift vector* which is given by,

$$\mathbf{u}_t^{(k)} = \mathbf{e}_t + \Delta\mathbf{v}_t^{(k)}.$$

It can be trivially proven that the weighted average of the drift vectors of the network is equal to the global statistics vector,

$$\frac{\sum_{k=1}^{n} w^{(k)}\mathbf{u}_t^{(k)}}{\sum_{k=1}^{n} w^{(k)}} = \overline{\mathbf{v}}_t$$

In other words, this is equivalent to the geometric property, that $\overline{\mathbf{v}}_t$ is in the convex hull of the drift vectors,

$$\overline{\mathbf{v}}_t \in \text{Conv}(\mathbf{u}_t^{(1)}, \mathbf{u}_t^{(2)}, \ldots, \mathbf{u}_t^{(n)}) \quad . \tag{4.1}$$

This property enables us to break down the global monitoring task into smaller, local tasks. Since individual nodes lack the capability to determine the convex hull of drift vectors, they must locally verify whether the monitored function f falls below or exceeds the threshold r in a local area of the $\mathbb{R}^d$. This region is given by a d-dimensional ball $B(\mathbf{e}_t, \mathbf{u}_t^{(k)})$ which is centered at $\frac{\mathbf{e}_t+\mathbf{u}_t^{(k)}}{2}$ and has a radius of $\left\|\frac{\mathbf{e}_t-\mathbf{u}_t^{(k)}}{2}\right\|$. It is known that,

$$\text{Conv}(\mathbf{u}_t^{(1)}, \mathbf{u}_t^{(2)}, \ldots, \mathbf{u}_t^{(n)}) \subset \bigcup_k B(\mathbf{e}_t, \mathbf{u}_t^{(k)})$$

$$\overset{4.1}{\Longrightarrow} \overline{\mathbf{v}}_t \in \bigcup_k B(\mathbf{e}_t, \mathbf{u}_t^{(k)})$$

Let's assign a color to each vector within the ball: red for vectors that meet the condition $\{\mathbf{x}|f(\mathbf{x}) < r\}$ and green for those where $\{\mathbf{y}|f(\mathbf{y}) \geq r\}$. A node must report its local statistics to the network when there are green vectors in the ball, indicating a breach of the local constraint. If the ball is monochromatically red, due to the relationship above, it is certain that $\overline{\mathbf{v}}_t < r$ and, thus, there is no need for communication.

In Figure 4.1, a visualization of drift vectors for 2-dimensional statistics is drawn on a plain for better understanding. All local statistics, at the start for $t = 0$, are located on the same point, at the blue square. This means that $\overline{\mathbf{v}}_0 = \mathbf{v}_0^{(k)}$, $\forall k \in [1, 5]$. The red dots are the local statistics at $t = t_1$ for all 5 nodes and the arrows pointing to them are the respective drift vectors. The gray area is the convex hull of the drift vectors and the circles are the balls $B(\mathbf{e}_{t_1}, \mathbf{u}_{t_1}^{(k)})$ of each node. At each point the individual nodes don't have information over the convex hull, but they can collectively monitor it using the union of circles. It is obvious that the union of the circles is a superset of the convex hull. Each node inspects its circle for constraint violations in order to determine whether or not it should share its statistics.

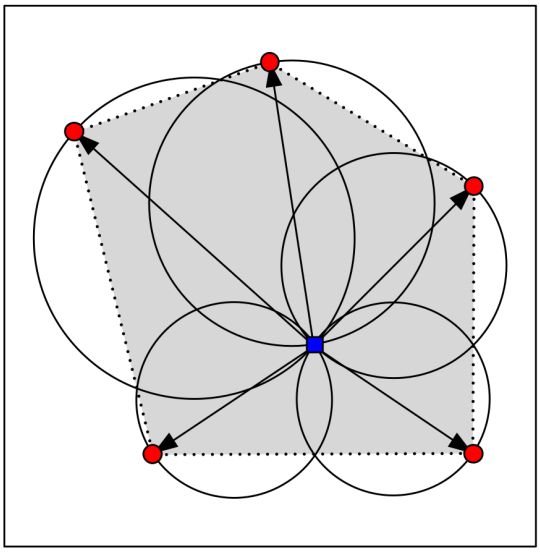


Figure 4.1: Drift Vectors Example
Source [35]

The decentralized geometric approach algorithm as presented in the paper [35] is displayed in Algorithm 11.

Algorithm 11 Geometric Approach Decentralized Algorithm [35]

Initially
Broadcast a message containing the initial statistics vector and update $\mathbf{v}'^{(k)}$ to hold the initial statistics vector. Upon receipt of messages from all the nodes, calculate the estimate vector $\mathbf{e}_t$.

Processing stage
Upon arrival of new data on the local stream, recalculate $\mathbf{v}_t^{(k)}$, and $\mathbf{u}_t^{(k)}$, and check if $B(\mathbf{e}_t, \mathbf{u}_t^{(k)})$ remains monochromatic. If not, broadcast the message $\langle k, \mathbf{v}_t^{(k)} \rangle$ and update $\mathbf{v}'^{(k)}$ to hold $\mathbf{v}_t^{(k)}$.
Upon receipt of a new message $\langle l, \mathbf{v}_t^{(l)} \rangle$, update $\mathbf{v}'^{(l)}$ to hold $\mathbf{v}_t^{(l)}$, recalculate $\mathbf{e}_t$, and check if $B(\mathbf{e}_t, \mathbf{u}_t^{(k)})$ is monochromatic. If $B(\mathbf{e}_t, \mathbf{u}_t^{(k)})$ is not monochromatic, broadcast the message $\langle k, \mathbf{v}_t^{(k)} \rangle$ and update $\mathbf{v}'^{(k)}$ to hold $\mathbf{v}_t^{(k)}$.

It should be evident at this point, that this approach can be implemented in a Distributed Machine Learning setting and may even extend to Federated Learning, as it offers a systematic strategy to minimize network communication. In this context, "nodes" can be thought of as individual learners or clients, "local and global statistics" as the parameters of a neural network, and "constraints" as local metrics for these parameters.

4.2 Functional Dynamic Averaging

Motivated by the Geometric Approach [35] and Functional Geometric Monitoring (FGM) [15, 32], V. Konidaris and V. Samoladas introduced a distributed machine learning synchronization method named *Functional Dynamic Averaging (FDA)* [33]. This method,

which includes three distinct monitoring techniques, significantly cuts down the cost of network communication. It surpasses comparable synchronization methods in actual training duration, while still maintaining high training accuracy.

FDA, being a synchronization method, can be implemented on top of any Synchronous DML algorithm like the one presented in Section 2.5.3. The standard Synchronous SGD, detailed in Algorithms 5 and 6, operates in communication rounds, each equal to one batch step in duration. It's a recognized principle in all DML algorithms that the global model's highest accuracy is attained when it closely approximates the average of all local models. Similarly, in FDA it is desired for the variance between the local weights of all clients $\mathbf{w}_t^{(k)}$ for k ranging from 1 to n and their mean model $\overline{\mathbf{w}}_t$, at step t, to be bellow a predetermined *threshold* Θ. Exceeding this threshold could lead to a trade-off with accuracy, which is not desired. The threshold Θ is considered a hyper-parameter of the FDA method that is defined at the beginning of the round and can be adjusted throughout the training process. Thus, the ideal condition for ending a communication round, known as the Round Termination Condition (RTC), can be mathematically expressed as,

$$\frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)}-\overline{\mathbf{w}}_t\right\|^2 \leq \Theta\ . \tag{4.2}$$

As long as the RTC holds we presume the individual learners haven't significantly deviated from their mean point. However, it is impossible to calculate the value of the mean model $\overline{\mathbf{w}}_t$ precisely, without exchanging the local weights at each training step. For this reason, the FDA method provides three different ways to approximately monitor the RTC.

In general, consider that there are n clients, each with its own local state $S_k(t)$ that belongs to an m-dimensional real space, denoted as $S_k(t) \in \mathbb{R}^m$ for $k = 1, \ldots, n$. The global state at time t, represented as $S(t)$, also lies in $\mathbb{R}^m$ and is defined as the average of all the local states. We aim to transform the RTC Inequality 4.2, into a new approximate monitoring formula. This formula will have the form $F(S(t)) \leq \Theta$, where $F : \mathbb{R}^m \to \mathbb{R}$ a non-linear function.

It is crucial to emphasize a significant point before moving to the three monitoring methods. Let there be t_0 the time the latest communication round started and therefore $\mathbf{w}_{t_0}^{(1)} = \mathbf{w}_{t_0}^{(2)} = \ldots = \mathbf{w}_{t_0}^{(n)}$. The *update* of learner k at step t is the difference of the weights at step t and the weights of the last synchronization t_0,

$$\boldsymbol{\Delta}_t^{(k)} = \mathbf{w}_t^{(k)} - \mathbf{w}_{t_0}^{(k)}\ .$$

As a result, the average update is given by,

$$\overline{\boldsymbol{\Delta}}_t = \frac{1}{n}\sum_{k=1}^{n}\boldsymbol{\Delta}_t^{(k)}\ .$$

Note that the left side of the RTC Inequality 4.2 can be written as,

$$\begin{aligned}
\frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)}-\overline{\mathbf{w}}_t\right\|^2 &= \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)}-\mathbf{w}_{t_0}^{(k)}-\left(\overline{\mathbf{w}}_t-\mathbf{w}_{t_0}^{(k)}\right)\right\|^2 \\
&= \frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}-\overline{\boldsymbol{\Delta}}_t\right\|^2 \\
&= \frac{1}{n}\sum_{k=1}^{n}\left(\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2-2\boldsymbol{\Delta}_t^{(k)}\cdot\overline{\boldsymbol{\Delta}}_t+\left\|\overline{\boldsymbol{\Delta}}_t\right\|^2\right) \\
&= \left(\frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2\right)-2\overline{\boldsymbol{\Delta}}_t\cdot\overline{\boldsymbol{\Delta}}_t+\left\|\overline{\boldsymbol{\Delta}}_t\right\|^2 \\
&= \left(\frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2\right)-\left\|\overline{\boldsymbol{\Delta}}_t\right\|^2
\end{aligned} \tag{4.3}$$

Thereby, we can conveniently define the local state of the k-th client and function F as,

$$S_k(t)=\begin{bmatrix}||\boldsymbol{\Delta}_t^{(k)}||^2\\ \boldsymbol{\Delta}_t^{(k)}\end{bmatrix}\in\mathbb{R}^{m+1}\text{ and }F\left(\begin{bmatrix}v\\ \mathbf{x}\end{bmatrix}\right)=v-||\mathbf{x}||^2\,, \tag{4.4}$$

and now due to 4.3 and 4.4 the RTC is equivalent to $F(S(t))\leq\Theta$.

However, defining the local state $S_k(t)$ in the described manner would be resource-intensive for the network. The local update $\boldsymbol{\Delta}_t^{(k)}$ has m dimensions, corresponding to the number of weights in the local client's model. Further down, our goal is to examine different ways to define the local states and the F function in order to minimize network overhead.

The FDA method can be used in both centralized and decentralized networks of clients. The general workflow in a centralized network is described by Algorithms 12 and 13 for any client k and the central server respectively.

Algorithm 12 Centralized Functional Dynamic Averaging Client k

repeat
  $\mathcal{B}\leftarrow$ (split $\mathcal{P}_k$ into batches of size $|B|$)
  Wait for server message m
  if $m=\langle\texttt{sync}\rangle$ **then** ▷ Server demands model synchronization
    Send $\mathbf{w}^{(k)}$ to central server
  else
    if $m=\langle\overline{\mathbf{w}}\rangle$ **then** ▷ First step of the new round
      $\mathbf{w}^{(k)}\leftarrow\overline{\mathbf{w}}$ ▷ Get the new updated global model
    end if
    $\mathbf{w}^{(k)}\leftarrow\mathbf{w}^{(k)}-\eta\cdot\sum_{x,y\in b}\frac{1}{|B|}\nabla_{\mathbf{w}}\ell(\mathbf{w}^{(k)};x,y)$ ▷ Where b the next batch from $\mathcal{B}$
    Send local state $S_k(t)$ to central server
  end if
until $m=\langle\texttt{term}\rangle$

Algorithm 13 Centralized Functional Dynamic Averaging Server

Initially $\overline{\mathbf{w}}$ = random values
  Broadcast message $m = \langle \overline{\mathbf{w}} \rangle$
repeat $(t = 0, 1, \ldots)$ ▷ Communication round
  Wait for $S_k(t)$ from all clients $k \in [1, n]$
  Compute global state $S(t) \leftarrow \frac{1}{n} \sum_{k=1}^{n} S_k(t)$
  if $F(S(t)) \leq \Theta$ **then** ▷ No synchronization
    Broadcast message $m = \langle \texttt{cont} \rangle$
  else ▷ Synchronization
    Broadcast message $m = \langle \texttt{sync} \rangle$
    Wait for $\mathbf{w}^{(k)}$ from all clients $k \in [1, n]$
    $\overline{\mathbf{w}} \leftarrow \frac{1}{n} \sum_{k=1}^{n} \mathbf{w}^{(k)}$
    Broadcast message $m = \langle \overline{\mathbf{w}} \rangle$
  end if
until stopping criterion is reached
Broadcast message $m = \langle \texttt{term} \rangle$ ▷ Stop training

Algorithm 14 Decentralized Functional Dynamic Averaging

Initially $\mathbf{w}^{(k)}$ = random values
  Broadcast $\mathbf{w}^{(k)}$
  Upon receipt of $\mathbf{w}^{(l)}$: $\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(l)}$
repeat
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size $|B|$)
  $\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k)} - \eta \cdot \sum_{x,y \in b} \frac{1}{|B|} \nabla_{\mathbf{w}} \ell(\mathbf{w}^{(k)}; x, y)$ ▷ Where b the next batch from $\mathcal{B}$
  Wait for $S_k(t)$ from all clients $k \in [1, n]$
  All reduce averaging: $S(t) \leftarrow \frac{1}{n} \sum_{k=1}^{n} S_k(t)$
  if $F(S(t)) > \Theta$ **then** ▷ Synchronization
    All reduce averaging: $\mathbf{w}^{(k)} \leftarrow \frac{1}{n} \sum_{k=1}^{n} \mathbf{w}^{(k)}$
  end if
until stopping criterion is reached ▷ Stop training

4.2.1 Naive FDA

The most trivial method to approximate monitoring of the RTC involves using the squared norm of the local update as the local state. By doing so, the dimensionality of the message to be communicated is reduced to one. Therefore, the local state at time t for client k and the function F can be expressed as,

$$S_k(t) = \left\| \boldsymbol{\Delta}_t^{(k)} \right\|^2 \in \mathbb{R} \quad \text{and} \quad F(v) = v \ .$$

It is easy to show these definitions satisfy the RTC.

$$
\begin{aligned}
F(S(t)) \leq \Theta &\Leftrightarrow \frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 \leq \Theta \\
&\Leftrightarrow \left(\frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2\right) - \left\|\overline{\boldsymbol{\Delta}}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 \leq \Theta \\
&\overset{4.3}{\Longleftrightarrow} \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 \leq \Theta \qquad (4.5) \\
&\Rightarrow \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \Theta \quad \underline{\text{RTC}}\ (4.2)
\end{aligned}
$$

The only drawback of using this *Naive* method of approximating the RTC is that, as shown in Inequality 4.5, sometimes it overestimates the true value of variance. In short, while the conditions for RTC are always met by the Naive FDA, the converse isn't always true. Fortunately, this doesn't happen often enough to pose any danger to the effectiveness of the algorithm. The methods analyzed below closer approximate the RTC by communicating more data at each step.

4.2.2 Linear FDA

In *Linear FDA* in addition to the squared norm we also send a reduced version of the local update. This is done by multiplication with a *unit vector* $\boldsymbol{\xi}$, so that a one-dimensional number results $\boldsymbol{\xi} \cdot \boldsymbol{\Delta}_t^{(k)} \in \mathbb{R}$. In such wise, the resulting states and function are,

$$
S_k(t) = \begin{bmatrix} ||\boldsymbol{\Delta}_t^{(k)}||^2 \\ \boldsymbol{\xi} \cdot \boldsymbol{\Delta}_t^{(k)} \end{bmatrix} \in \mathbb{R}^2 \text{ and } F\left(\begin{bmatrix} v \\ x \end{bmatrix}\right) = v - x^2
$$

Again, it is easily proven that Linear FDA satisfies the RTC.

$$
\begin{aligned}
F(S(t)) \leq \Theta &\Leftrightarrow \frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 - \frac{1}{n}\sum_{k=1}^{n}(\boldsymbol{\xi} \cdot \boldsymbol{\Delta}_t^{(k)})^2 \leq \Theta \\
&\Leftrightarrow \left(\frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2\right) - \left\|\overline{\boldsymbol{\Delta}}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 - (\boldsymbol{\xi} \cdot \overline{\boldsymbol{\Delta}}_t)^2 \leq \Theta \\
&\overset{4.3}{\Longleftrightarrow} \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 - (\boldsymbol{\xi} \cdot \overline{\boldsymbol{\Delta}}_t)^2 \leq \Theta \\
&\Rightarrow \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \Theta \quad \underline{\text{RTC}}\ (4.2)
\end{aligned}
$$

Randomly choosing the unit vector $\boldsymbol{\xi}$ is proven to be suboptimal, as it would most likely be orthogonal to the average update $\overline{\boldsymbol{\Delta}}_t$. The ideal option for $\boldsymbol{\xi}$ would be for it to be linked with $\overline{\boldsymbol{\Delta}}_t$, but this implementation would be impractical. Instead we create vector $\boldsymbol{\xi}$ in accordance to $\overline{\boldsymbol{\Delta}}_{t_0}$, i.e., the average update of the last client synchronization.

This vector can be easily computed without communication, by also locally storing $\overline{\mathbf{w}}_{t_{-1}}$, the weights of the next to last synchronization. By this, all clients can locally compute $\overline{\boldsymbol{\Delta}}_{t_0} = \overline{\mathbf{w}}_{t_0} - \overline{\mathbf{w}}_{t_{-1}}$.

4.2.3 Sketch FDA

The *Sketch FDA* method utilizes the concept of *AMS sketches* [9] to estimate the average update. The AMS sketch, with notation $\text{sk}(\cdot)$, is a function that compresses a large vector into a much smaller matrix. Let a vector $\mathbf{v} \in \mathbb{R}^M$, then its sketch is $\text{sk}(\mathbf{v}) \in \mathbb{R}^{m \times d}$ given by,

$$\text{sk}(\mathbf{v}) = \boldsymbol{\Xi} = \begin{bmatrix} \boldsymbol{\xi}_1 & \boldsymbol{\xi}_2 & \dots & \boldsymbol{\xi}_d \end{bmatrix} \text{, where } d \cdot m \ll M.$$

The sketch function is linear and can be computed in $\mathcal{O}(dM)$ steps. There also exists a function, notated as $\mathcal{M}_2$, that can quite accurately estimate the squared norm of a vector $\mathbf{v}$ using its sketch $\text{sk}(\mathbf{v})$ as input. The function is,

$$\mathcal{M}_2(\text{sk}(\mathbf{v})) = \underset{i=1,\dots,d}{\text{median}} \|\boldsymbol{\xi}_i\|^2 .$$

For $m = \mathcal{O}\left(\frac{1}{\epsilon^2}\right)$ and $d = \mathcal{O}\left(\log \frac{1}{\delta}\right)$ it is proven that with probability at least $1-\delta$ we can estimate the value of $\|\mathbf{v}\|^2$ in the range,

$$(1-\epsilon)\|\mathbf{v}\|^2 \leq \mathcal{M}_2(\text{sk}(\mathbf{v})) \leq (1+\epsilon)\|\mathbf{v}\|^2 \ . \tag{4.6}$$

In this case, the large vector that needs compression is the local update $\boldsymbol{\Delta}_t^{(k)} \in \mathbb{R}^M$. On that account, the local state and F function for Sketch FDA are defined as,

$$S_k(t) = \begin{bmatrix} ||\boldsymbol{\Delta}_t^{(k)}||^2 \\ \text{sk}\left(\boldsymbol{\Delta}_t^{(k)}\right) \end{bmatrix} \in \mathbb{R}^{1+d\times m} \text{ and } F\left(\begin{bmatrix} v \\ \boldsymbol{\Xi} \end{bmatrix}\right) = v - \frac{1}{1+\epsilon}\mathcal{M}_2(\boldsymbol{\Xi}) \ .$$

Now, to prove that the RTC implies,

$$\begin{aligned}
F(S(t)) \leq \Theta \Leftrightarrow & \frac{1}{n}\sum_{i=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 - \frac{1}{1+\epsilon}\cdot\frac{1}{n}\sum_{i=1}^{n}\mathcal{M}_2\left(\text{sk}\left(\boldsymbol{\Delta}_t^{(k)}\right)\right) \leq \Theta \\
\overset{linearity}{\Longleftrightarrow} & \frac{1}{n}\sum_{i=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 - \frac{1}{1+\epsilon}\mathcal{M}_2\left(\text{sk}\left(\overline{\boldsymbol{\Delta}}_t\right)\right) \leq \Theta \\
\overset{4.6}{\Longleftrightarrow} & \left(\frac{1}{n}\sum_{i=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2\right) - \left\|\overline{\boldsymbol{\Delta}}_t\right\|^2 \leq \frac{1}{n}\sum_{i=1}^{n}\left\|\boldsymbol{\Delta}_t^{(k)}\right\|^2 - \frac{1}{1+\epsilon}\mathcal{M}_2\left(\text{sk}\left(\overline{\boldsymbol{\Delta}}_t\right)\right) \leq \Theta \\
\overset{4.3}{\Longrightarrow} & \frac{1}{n}\sum_{i=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \Theta \quad \underline{\text{RTC}} \ (4.2)
\end{aligned}$$

4.3 KungFu Adaptation

KungFu is open for customization as it prioritizes custom user-designed Adaptation Policies (APs). Also, explicit training loop implementation is encouraged so that custom

training functions can be called in each iteration.

Our implementation is based, as well, on a custom training loop using the `GradientTape` TensorFlow class. `GradientTape` records the operations that happen during a forward pass and then uses them to calculate the local gradients at every step. We have opted out of using the wrapper optimizers developed by the KungFu team. This decision was reached as we intended to implement the Functional Dynamic Averaging (FDA) methods, as described in Section 4.2. The FDA methods aggregate model weights, while the KungFu optimizers aggregate gradients.

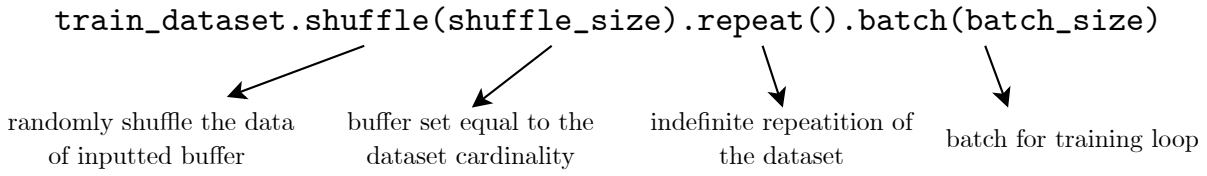
We have also opted to use eager execution of TensorFlow functions as the FDA methods require if conditions. So, graph execution usage is limited.

The training loop logic is common for all experiments and inside the `tf2_mnist_experiment.py` file. The standard local training loop is confined in a function that uses `GradientTape`, and it gets wrapped around FDA functions that call it. At the end of each epoch, a model evaluation happens using the model of the client with `current_rank()=0`. For FDA methods, we evaluate the last model derived from a synchronization, to avoid overfitting with local data.

We have used three distinct FDA Python functions to reference three Python files — located in folder `/fda_functions`— that contain the FDA logic. The FDA logic contains squared norm, unit vector and AMS sketch calculation functions. Only when the RTC gets violated, the FDA functions perform an all-reduce operation to aggregate all clients weights and therefore assign the aggregated weights to local models. The last and second to last synchronization models, necessary to approximate the RTC, are stored in the experiment Python file as global variables.

Regarding the dataset, our choice was the *MNIST dataset* [12]. MNIST is a dataset containing a large number of handwritten digits in the form of grayscale 28×28 pixel images. It has become the standard dataset for image recognition, often termed the "Hello World" of Deep Learning and Optical Character Recognition (OCR). The training dataset contains 60,000 images and the test dataset 10,000. Since we have expected that our experiments would last multiple epochs, we had to design the training dataset so that clients wouldn't run out of data. First, there was a need to normalize the data and shard them using the TensorFlow method `shard(N,i)`, where `N` is the number of clients and `i` the current client. This is important so that the dataset gets partitioned and each client gets its own separate share of it.

The shards of the dataset have to get repeated in order to train for multiple epochs, without exchanging local data. The TensorFlow methods used for dataset manipulation are explained below.



The experiments needed to run in a High-Performance Computing (HPC) setting using the Slurm Workload Manager. Most HPC infrastructure serve a multitude of users

and for that reason resources are extremely valuable. Individual jobs submitted to the server enter a priority queue when resources are limited. Therefore, we needed to reduce the number of jobs submitted to the server to the absolute minimum, and automate the process. This is achieved by describing each experiment's hyper-parameters in a JSON file format. Python script `experiments_in_json.py` receives as input a list of hyper-parameters possible values and creates a JSON file with all possible experiments as JSON objects. Then, the `job_submitter.py` script creates a custom Slurm job script based on the inputted values (see Figure 4.2).

The Slurm job reads the IP addresses of the participating HPC nodes and calls the `run_experiments.py` script. This script is responsible for running in sequence all KungFu experiments, described in the JSON file, using the `subprocess` Python library. In between individual KungFu command calls (experiments), all nodes run a "`sleep 1m`" command, waiting for all other nodes to end their previous experiment.

During every experiment a Python dictionary stores important data results. The results are of three types; step, epoch and info results. Info are general information of the experiment concerning hyper-parameters and overall results. Step results are collected at each step and include loss, time and whether a synchronization occurred. Epoch results also include accuracy evaluation data. Each of these three types is collected in a Pandas [26] dataframe in the `logs_df.py` script. The info dataframe is appended in the `info.csv` file where all experiments information is stored and it has its own unique experiment ID. Each experiment stores all its step and epoch results in CSV files, but this time each one has its own file.

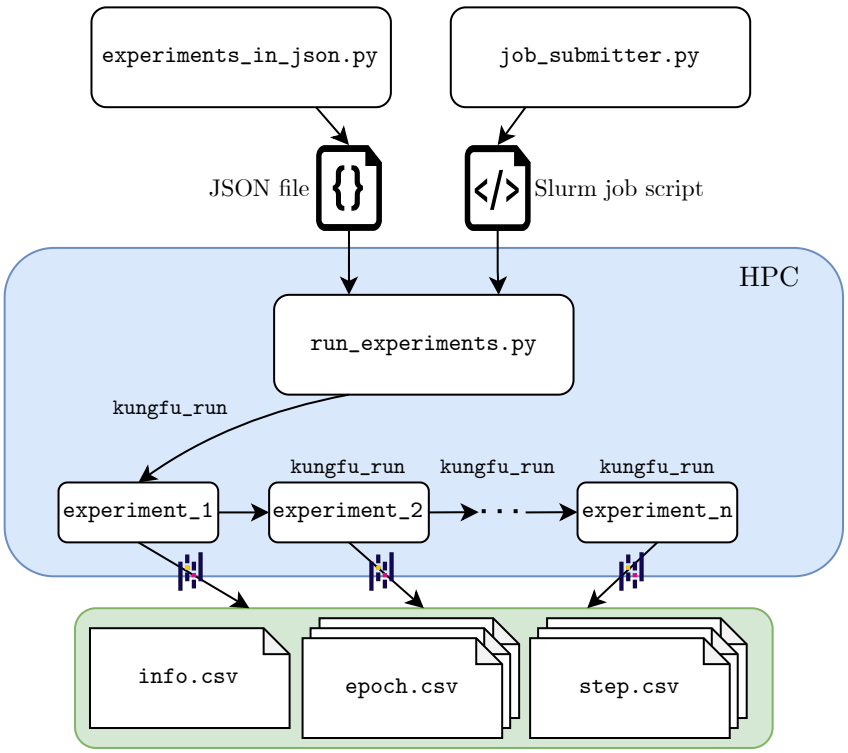


Figure 4.2: Workflow of Slurm Job

Chapter 5

Experiments and Results

In this chapter, we will explain in detail the preliminary work that was needed to perform the experiments and also present and analyze the results that were generated. Preliminary work involves code structure, training details, KungFu customization and data analysis description.

5.1 Preliminary Work

5.1.1 Experimental Setup and Infrastructure

The experiments were performed using the KungFu distributed machine learning library, q.v. Section 3.3. The source code was forked from the project's GitHub repository and customized in order to facilitate our needs. At first, it was crucial to design a comprehensive and effortless way to install KungFu on the available hardware, using the available source code.

The initial experiments were performed on a personal computer environment of only one node and moved on to cluster and High-Performance Computing (HPC) infrastructure. The first attempt on a cluster was the CPU "Grid" Cluster at Technical University of Crete (TUC), however the CPU instruction set required from KungFu wasn't available to this server.

A more modern and customizable option was used afterwards and namely the "~okeanos-knossos" cloud IaaS service provided by the National Infrastructures for Research and Technology (GRNET). The ~okeanos-knossos is equipped with a convenient user interface to create, edit and destroy custom CPU-powered virtual machines (VMs). The use of ~okeanos-knossos, with its simple VM architecture, was pivotal assisting in the understanding of KungFu's function in a distributed environment.

On this server, a custom Conda environment [2] was designed so that the installation of KungFu on a multitude of VMs can be simpler and speedier. Conda environments are collections of software packages that complement or replace the default packages installed on a machine. These package collections can be described in `.yaml` files, so they can be used in different machines without the effort of package reinstallation. The package `go-cgo` was the most important for building the project as it was the only one who managed to compile the communication layer library designed by KungFu.

Further experimentation, involving GPUs this time, was conducted using the newly-assembled Polytechnix GPU server at TUC. Nonetheless, hardware on this server was insufficient for the purpose of the ultimate experiments. The experiments demanded a GPU cluster consisting of multiple nodes, and that ideally involves a handy scheduler. For that reason, the final solution would be to run the experiments on the ARIS HPC supercomputer from GRNET S.A. [3]. The ARIS supercomputer provides 44 GPU accelerator nodes each containing 2 processors. ARIS admits projects from any Greek educational institution and thus a variety of users and purposes.

This workload needs to somehow get balanced by a central service. Consequently, ARIS employs the Slurm workload manager [39] to manage submitted jobs. Users are obligated to describe their job requirements in a Bash file, i.e. number of nodes, RAM usage, wall time. If all nodes are allocated, Slurm adds the job in a queue, with its rank depending on its requirements.

5.1.2 Model

The model we needed to select for the experiments had to be powerful enough to achieve high accuracy, comparable to state of the art performance. This way we could examine if the Geometric Approach algorithms can maintain high accuracy values, while reducing training time. In order to keep the number of variables in the neural network low, we had to use a Convolutional Neural Network (CNN).

The network has been developed by Michail Theologitis from the Technical University of Crete for his own Federated Learning project and termed "Advanced CNN" (Figure 5.1). The network uses multiple convolutional layers, with ReLU as the activation function, and multiple pull layers. It consists of 2,592,202 variables in total.

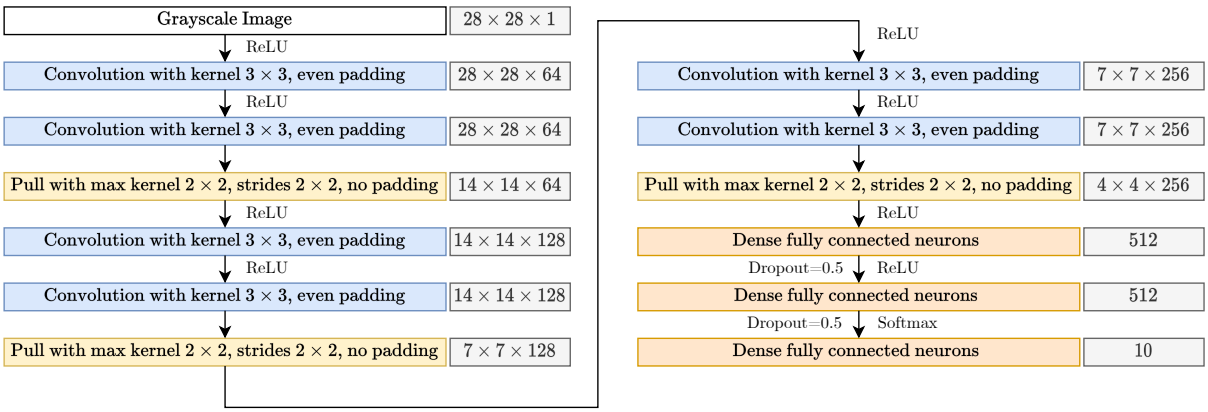


Figure 5.1: Advanced CNN

5.2 Experiments

Our main focus regarding the experiments was to show how the Geometric Approach —using the Functional Dynamic Averaging (FDA) methods— perform in various hyper-parameters settings. In order to isolate each hyper-parameter and observe the changes

that each of them induces to a number of metrics, it was necessary to arbitrarily choose a set of default hyper-parameter values. Each parameter examined alters its value based on a set described by us, while all other parameters remain stable on their default values.

The FDA methods are compared to a baseline synchronous algorithm, that essentially aggregates model variables for every training step. It is equivalent to training with zero threshold, minus the local state computation and communication.

The parameters that where used are the following. They include hyper-parameters for number of clients, batch size and threshold. The values in **bold** font represent the default values.

Repeat 3 times {

1. Optimizer: Adam (learning rate 10^{-3})
2. Epochs: 50
3. Model: Advanced CNN
4. Algorithms: Synchronous, Naive FDA, Linear FDA, Sketch FDA
5. Number of Clients: 4, 8, **16**, 32
6. Batch Size: 64, **128**, 256
7. Threshold: 1, 25, **50**, 100, 200

The number of total experiments performed can be extracted by the following calculation,

$$(3 \times (7 \text{ FDA exp.}) + (3 \text{ Synchronous exp.})) \times (4 \text{ clients setups}) \times (3 \text{ repetitions}) = 288.$$

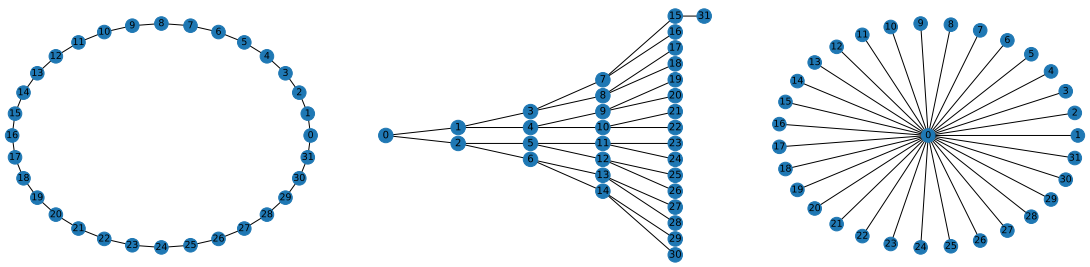
Hopefully, an automated mechanism to execute 288 experiments was developed, thoroughly analyzed in Section 4.3. This mechanism provided the advantage to concentrate all 288 experiments to only four Slurm [39] jobs, as many as the number of different clients setups.

To assess the efficacy of the Geometric Approach using a broader range of network topologies, six extra experiments have been executed. The earlier experiments have been using the default ring topology of KungFu, but this topology isn't common in Federated Learning scenarios. Ring topologies are beneficial to cluster-based training. Since the Geometric Approach was designed with decentralized learners in mind, it was reasonable to assume that they would perform significantly better in these topologies compared to the baseline algorithm. As the results would be more evident in a bigger network, it was decided to execute them in a network of 32 clients. The parameters of the experiments were:

1. Optimizer: Adam (learning rate 10^{-3})
2. Epochs: 50
3. Model: Advanced CNN

4. Algorithms: Synchronous, Naive FDA, Linear FDA, Sketch FDA
5. Number of Clients: 32
6. Batch Size: 128
7. Threshold: 2

The default ring topology (Figure 5.2a) used by KungFu creates an optimal ring based on the topology of the GPU nodes in the network as shown in Figure 3.3. The topologies that we investigated were a *star* and *binary tree* topology. The star topology would be the most wearing to the network as all peer-to-peer communication would require communication with central node "0". On the other hand, the binary tree topology would be an in-between solution as it is a complete tree and distributes network overhead in equal shares.



(a) Ring (b) Binary Tree (c) Star

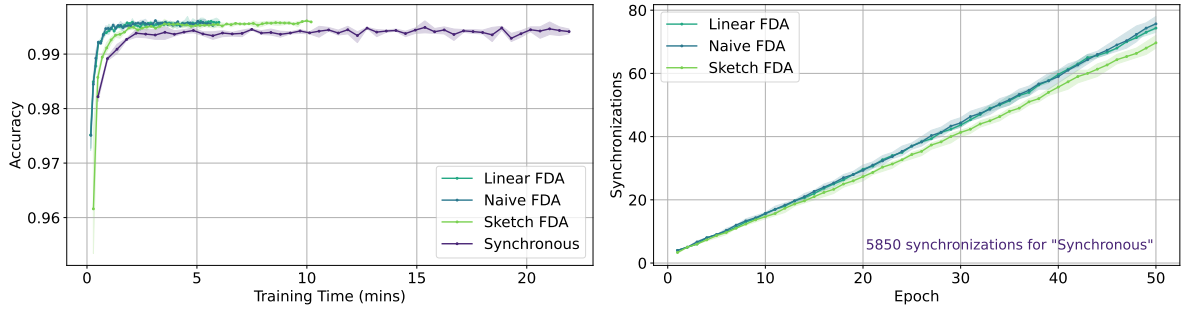
Figure 5.2: Network Topologies

These experiments, as discussed earlier, were executed using the ARIS HPC [3] [1]. The GPU partition of ARIS provides 44 nodes each containing two *Haswell - Intel(R) Xeon(R) E5-2660v3* processors. KungFu has the ability to add multiple slots for clients in a single node and thus a decision had to be made, on how many clients would exist on each node. Having two GPU processors on each node we concluded to use two clients per node, although this would slightly limit inter-GPU communication. So, from now on two clients are synonymous to one GPU node.

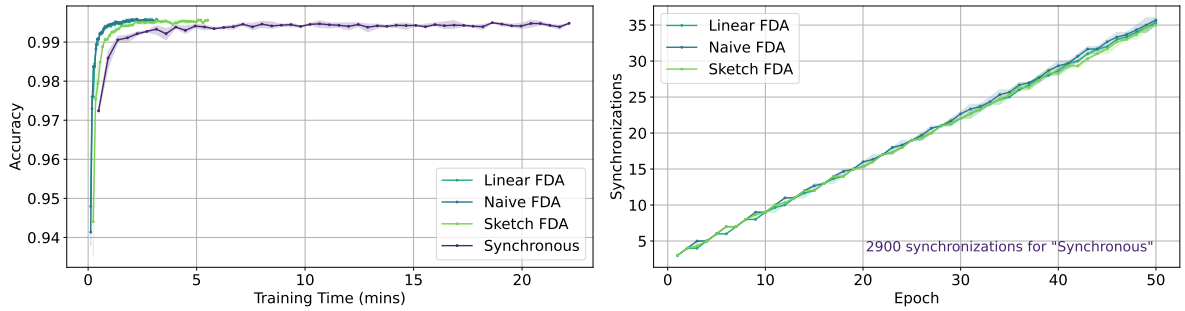
5.3 Results

The visualization of the experiments' results was possible with the use of a data analysis Python class that reads CSV files and registers their data in three Pandas [26] dataframes as the ones created during experiment execution (q.v. Figure 4.2). This class is used by a Jupyter notebook dedicated to data visualization. The main points of focus are accuracy, number of synchronizations (a.k.a. communication rounds) and execution time.

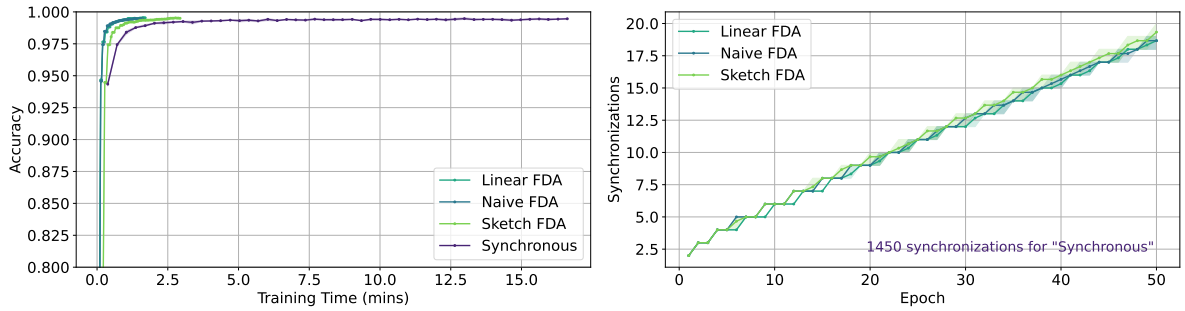
[1]This work was supported by computational time granted from the National Infrastructures for Research and Technology S.A. (GRNET S.A.) in the National HPC facility - ARIS - under project ID pa230902



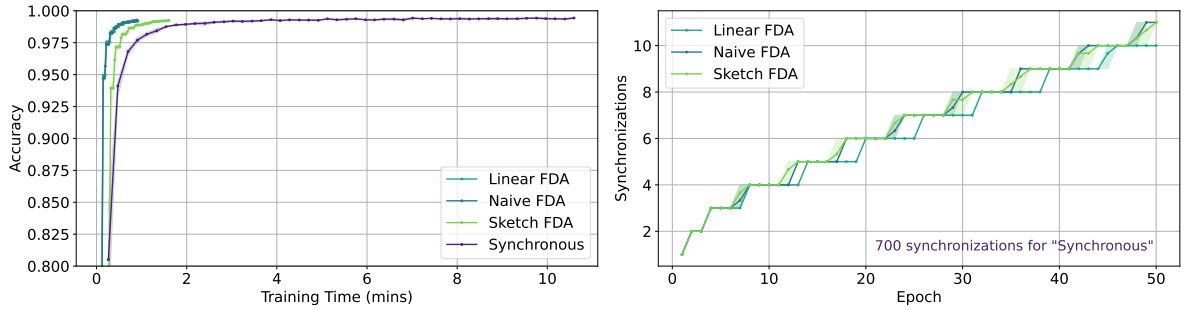
(a) 4 Clients



(b) 8 Clients



(c) 16 Clients

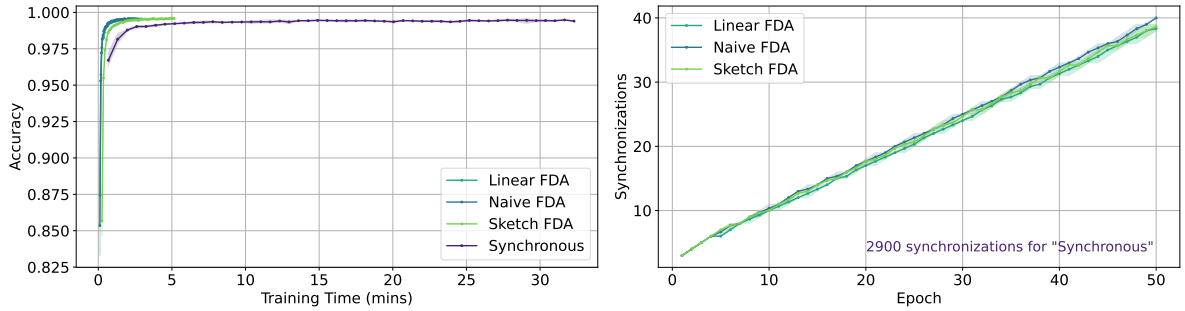


(d) 32 Clients

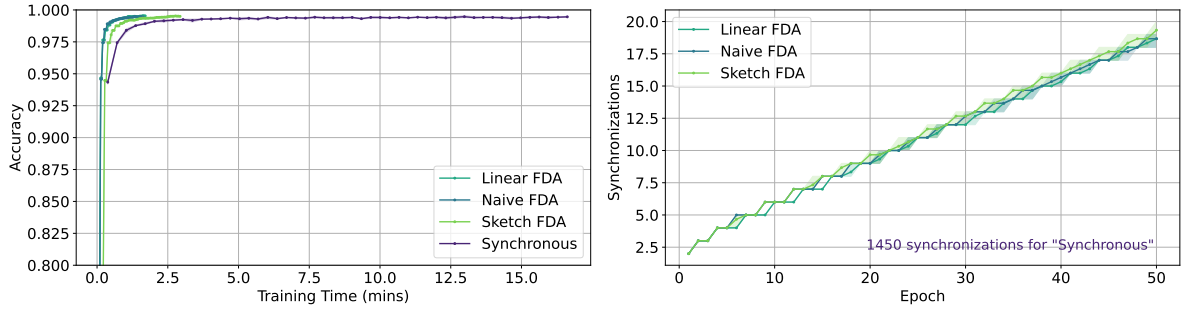
Figure 5.3: Accuracy and synchronizations metrics with variable number of clients

As the number of clients increases, the variance between the local models and their average is reducing and thus fewer synchronizations are performed. The FDA methods outperform the baseline algorithm in terms of training duration and accuracy. The duration advantage of the FDA methods becomes more prominent with the increase of clients in the network as communication becomes more and more expensive. For the detailed

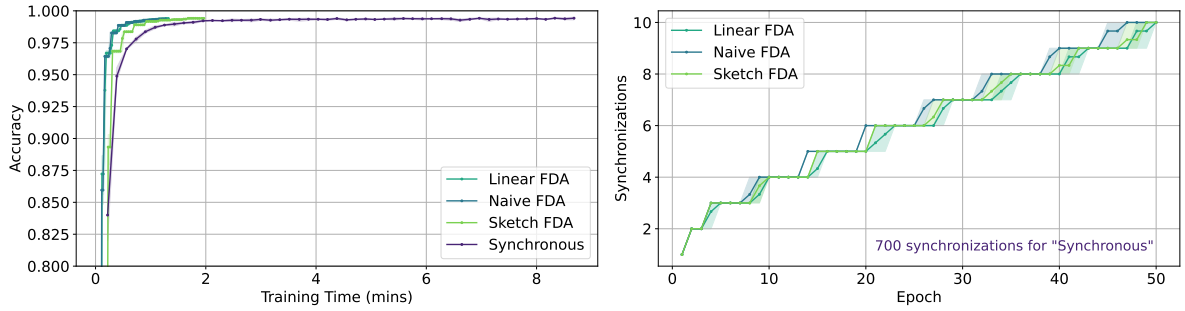
training duration results refer to the Appendices' Table A.1. In real time the FDA methods are equally superior to Synchronous in terms of accuracy as they repeatedly reach high accuracy prematurely.



(a) Batch Size 64



(b) Batch Size 128

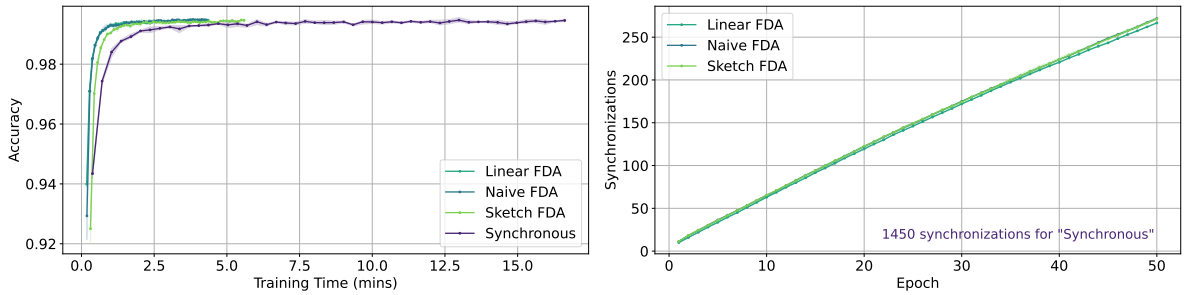


(c) Batch Size 256

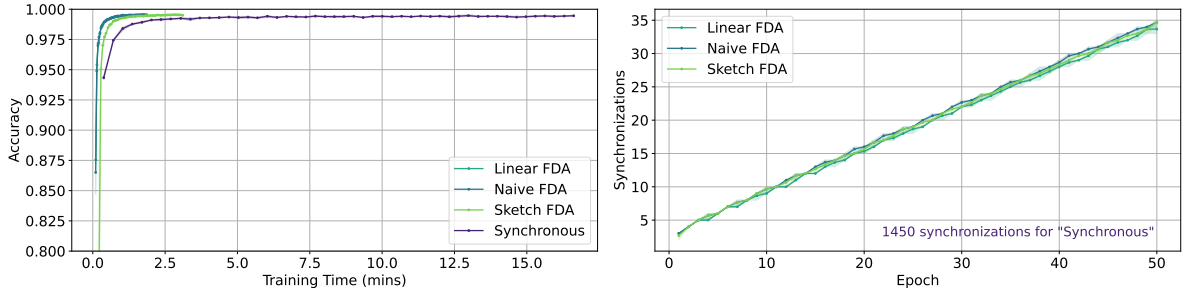
Figure 5.4: Accuracy and synchronizations metrics with variable batch size

The batch size like the number of clients reduces the number of synchronizations. Training steps are becoming larger and consequently each epoch has fewer steps. The number of synchronizations is inversely proportional to the batch size. Increasing the batch size by two, cuts synchronizations in half.

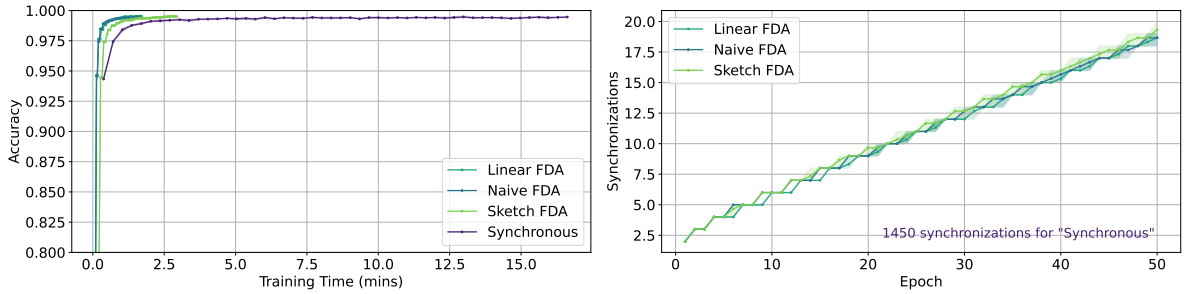
Although the number of communication rounds decreases for larger batch sizes, the total training duration of the FDA Methods approaches more the duration of the Synchronous algorithm. This is because the Synchronous algorithm benefits more from the decreasing number of total steps. For the detailed training duration results see Table A.2. The same figures for 4, 8 and 32 clients are appended in Figures A.1, A.2 and A.3.



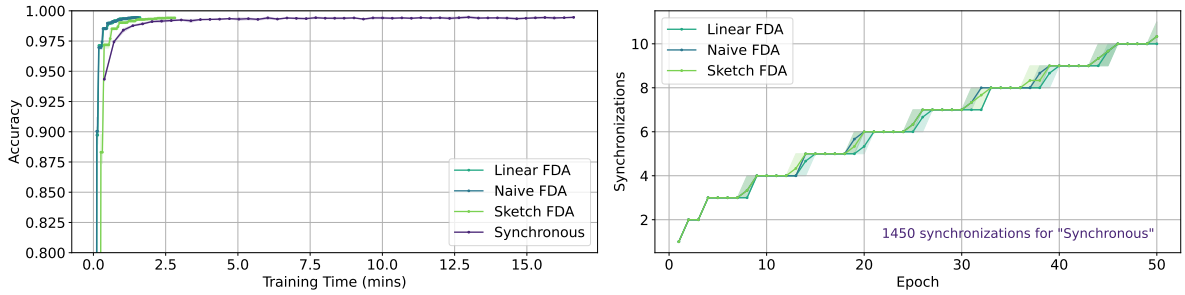
(a) Threshold 1



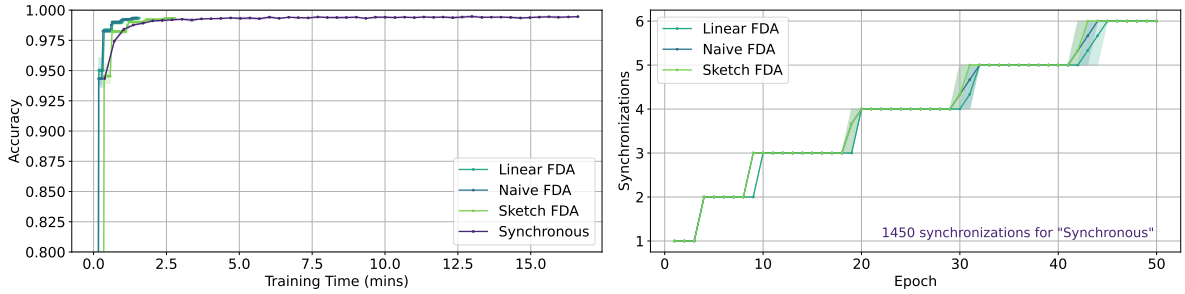
(b) Threshold 25



(c) Threshold 50



(d) Threshold 100



(e) Threshold 200

Figure 5.5: Accuracy and synchronizations metrics with variable threshold

In Figure 5.5 accuracy and synchronizations results are shown with variable threshold. The smaller the threshold value the more the FDA methods emulate the performance of the Synchronous algorithm. They increase their accuracy quickly in the first few epochs and their graphs oscillate heavily. This is because the number of communication rounds is increasing. A larger threshold value means limited communication and decreased training duration, but at the expense of accuracy, especially in the first ten epochs. Nonetheless, still they outperform Synchronous in terms of accuracy, even for large threshold, but only for large numbers of clients.

Training duration results can be found in Table A.3 and the equivalent figures for 4, 8 and 32 clients in Figures A.4, A.5 and A.6. In Figures A.4 and A.5 and for large threshold values Synchronous has comparable accuracy performance.

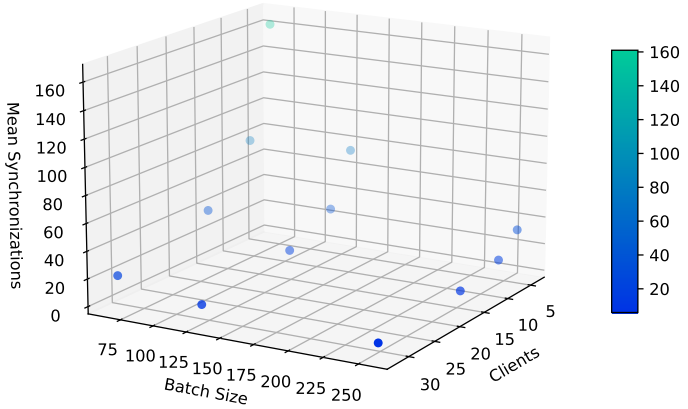
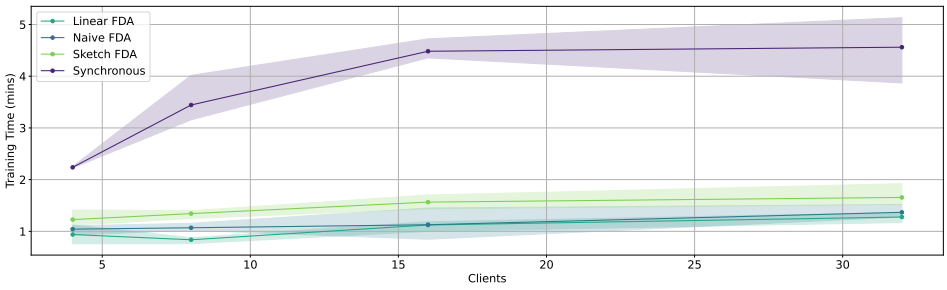
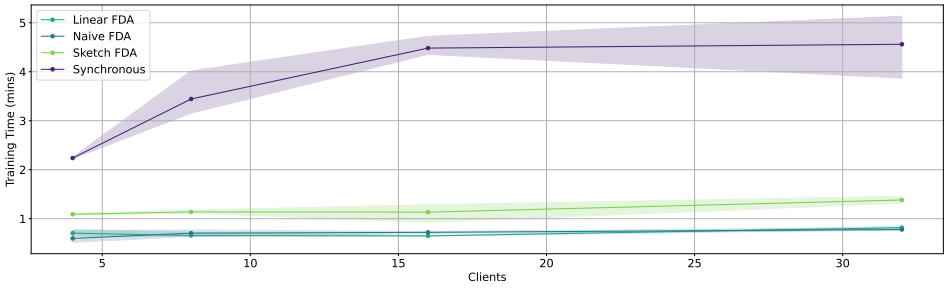


Figure 5.6: Synchronizations with variable number of clients and batch size



(a) Threshold 1



(b) Threshold 25

Figure 5.7: Training time until accuracy of 99.3% is reached over the number of clients

Figure 5.6 shows the mean number of synchronizations for the default threshold value 50. It is clear that the number of communication rounds fluctuates despite the stable threshold value. The general pattern is that for a higher number of clients and larger batch size the number of communication rounds decreases.

We can also observe that the number of synchronizations is equal for different experiments. For example, experiments with 4, 8, 16 clients and batch size 256, 128, 64 correspondingly all have the same number of synchronizations. This can be explained by the fact that the "global batch size" in those experiments stays the same, $4 \times 256 = 8 \times 128 = 16 \times 64 = 1024$.

In Figure 5.7 we show the training time needed to reach an accuracy of 99.3% by the several algorithms, in relation to the number of clients. We show results for the default batch size 128 and for threshold values 1 and 25. This is deliberate, as for higher threshold values and 32 clients, this accuracy value cannot be reached in just 50 epochs. This is another indication that not all threshold values are suitable for all scenarios. The difference in training time between the Synchronous and the FDA methods is pronounced, with the Sketch FDA method lagging a bit behind compared to Naive and Linear. What is quite worrying is that all algorithms record higher training time as the number of clients increases, although we would expect the higher computational parallelization to reduce it. This might mean that the MNIST dataset isn't complicated enough to justify a large number of clients.

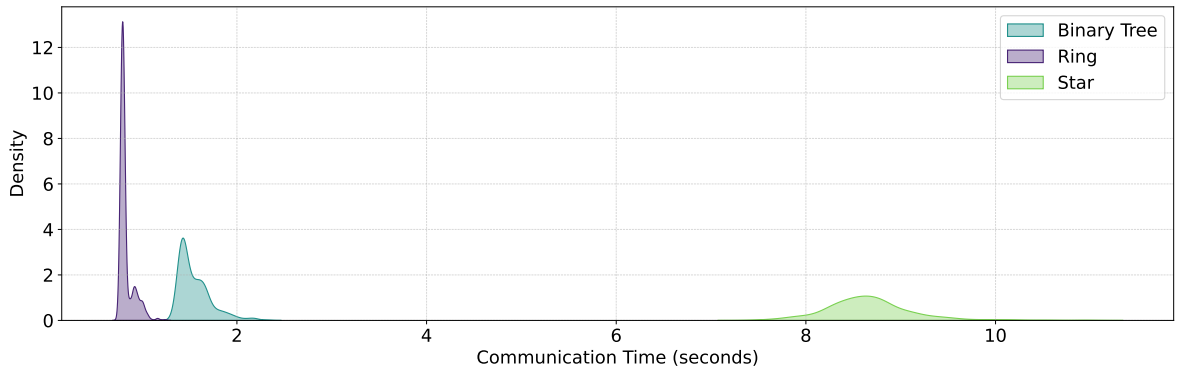
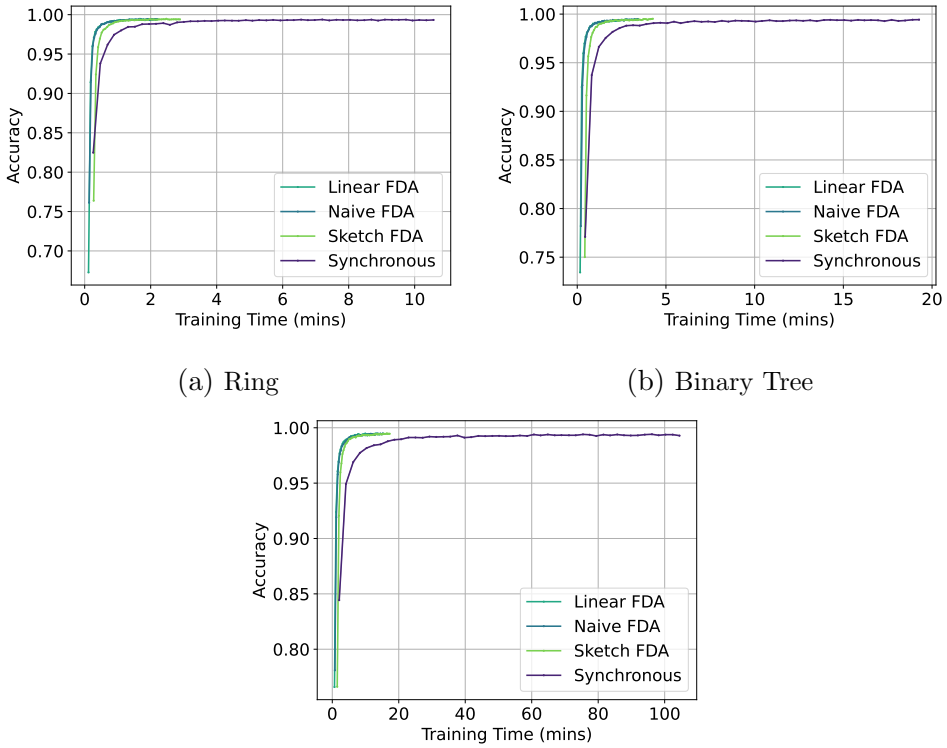


Figure 5.8: Communication time distribution with variable topology

Moving on to the topology experiments, we have created a density graph for the communication time of each synchronization in Figure 5.8. The data used on this figure are taken from the Synchronous algorithm experiment as it had, by nature, many more synchronization data. So for this example, the measured communication time is the time needed to perform the all-reduce operation. We can evidently detect that the default ring topology is the optimal out of the three, with the binary tree topology closely lagging behind and the star topology being considerably slower.



(a) Ring

(b) Binary Tree

(c) Star

Figure 5.9: Accuracy with variable topology

Having in mind that communication is more expensive for the new network topologies, we assumed that the training performance of the FDA methods would be even more beneficial. This is clear in Figure 5.9, as the FDA methods perform much better in real training time for 50 epochs. While the FDA methods are around five times faster than the baseline algorithm for the default topology, they are around six and seven times faster for the binary tree and star topologies. This speed up happens without any accuracy trade-off.

Chapter 6

Conclusions

The current thesis aims to develop a deployable Federated Learning tool using Tensorflow, that is based on the theoretic framework of the Geometric Approach. We have selected to extend the KungFu distributed learning library as it is offering state of the art throughput speeds and provides adaptive mechanisms especially on custom network topologies.

The algorithms that we have chosen to implement were the three variations of Functional Dynamic Averaging (FDA), termed Naive, Linear and Sketch. We have verified what was already proven in theory. The FDA methods outperform a classic synchronous algorithm in a vast set of hyper-parameters. These positive results have been even more pronounced when executing in suboptimal network topologies that simulate real-life decentralized networks of distant nodes. This is what convinces us that this work would have direct application on Federated Learning scenarios, where individual nodes communicate through poor internet connections.

Something that stands out in the results is that the arbitrary value set as a threshold in the FDA methods does not fit all hyper-parameter configurations. To make training more adaptable, we could develop a functional solution to automatically set the threshold value based on the number of clients and the batch size and aiming for a stable number of synchronizations. This could save valuable time spent on threshold tuning. Regression models and exhaustive experimentation could be the solution to this problem.

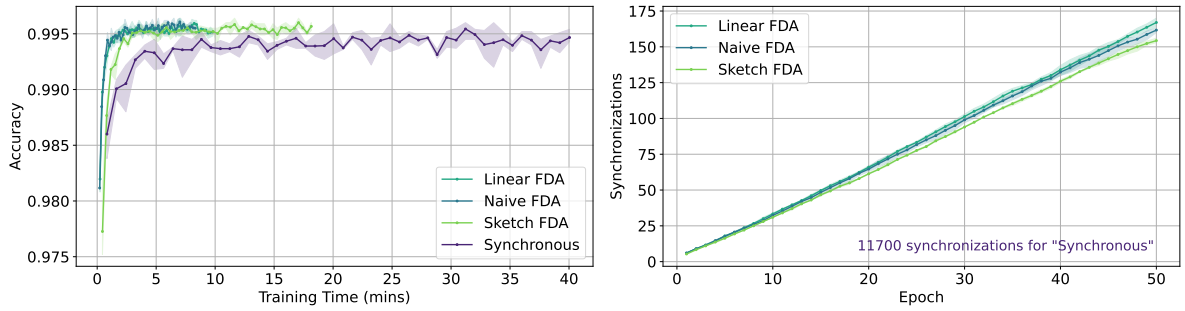
Although the work that have been done is significant, it is still a small portion of the available experimentation choices. We haven't experimented with the use of other optimizers. Adam can be widely used in the field, but it is not the only one and isn't suitable for all Machine Learning cases.

It has been observed that the synchronous baseline algorithm used requires more time to achieve an accuracy goal as the number of nodes increases. This means that the complexity and duration of the training process is miniscule compared to the communication overload. This is an indication that we have reached the limits of the MNIST dataset, which may not offer enough complexity to justify the use of large networks of learners. Consequently, we would like to test our work on more complex and demanding datasets to extract more variable results.

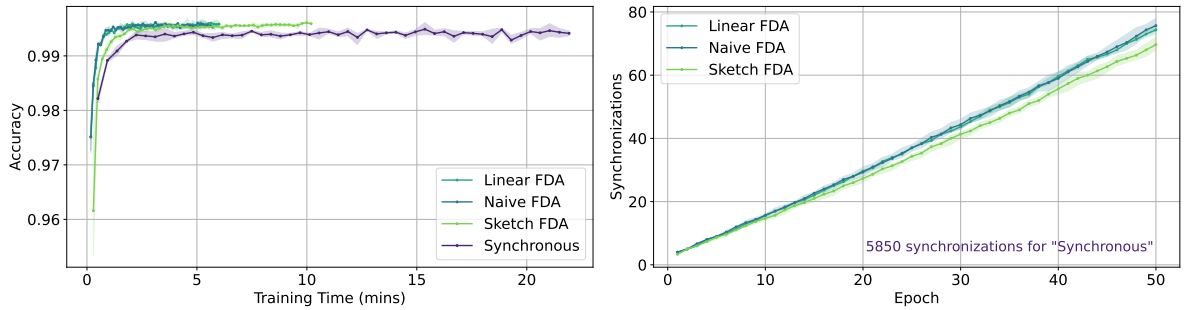
Appendices

Appendix A

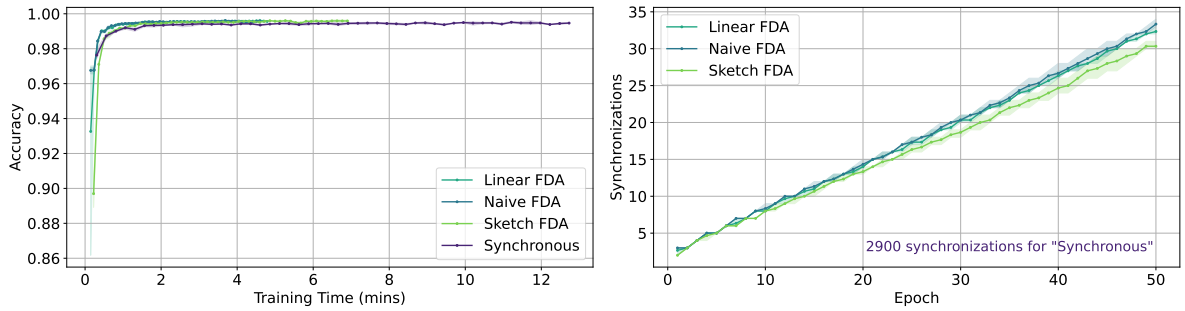
Additional Results



(a) Batch Size 64

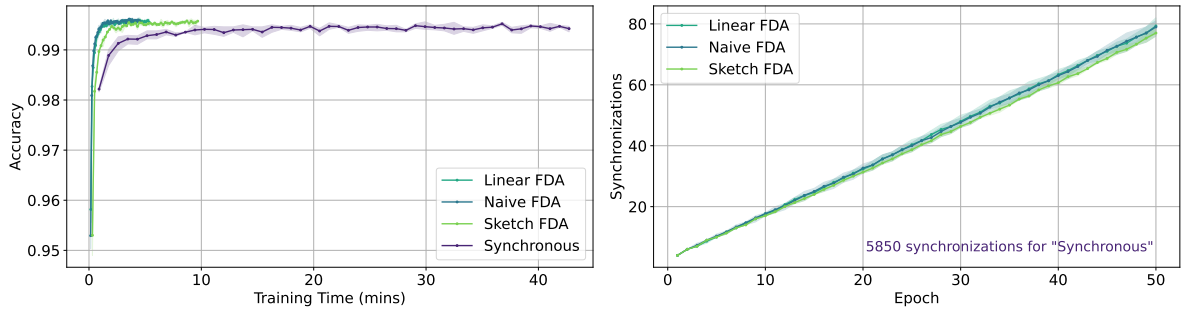


(b) Batch Size 128

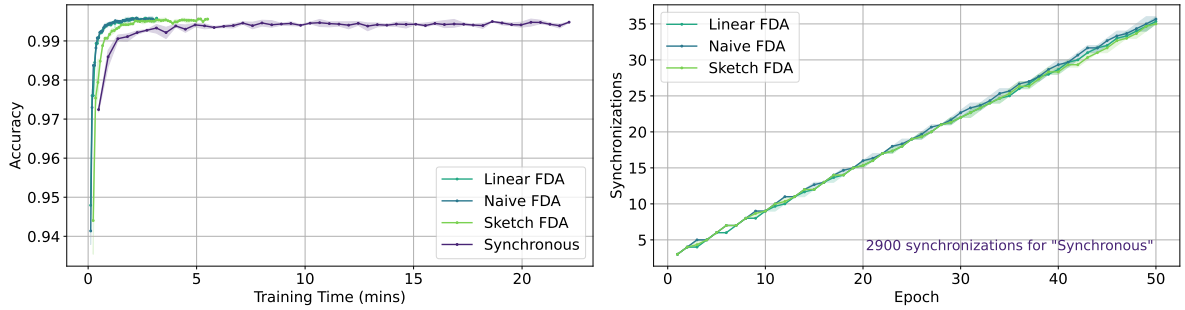


(c) Batch Size 256

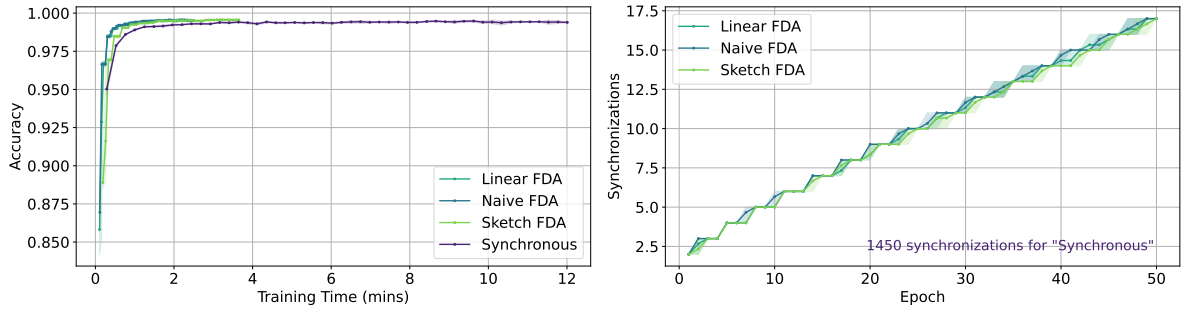
Figure A.1: Accuracy and synchronizations metrics with variable batch size (4 Clients)



(a) Batch Size 64

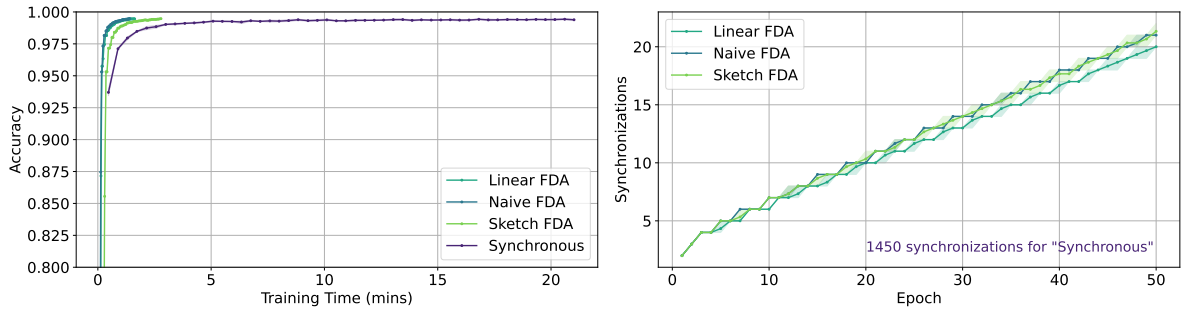


(b) Batch Size 128

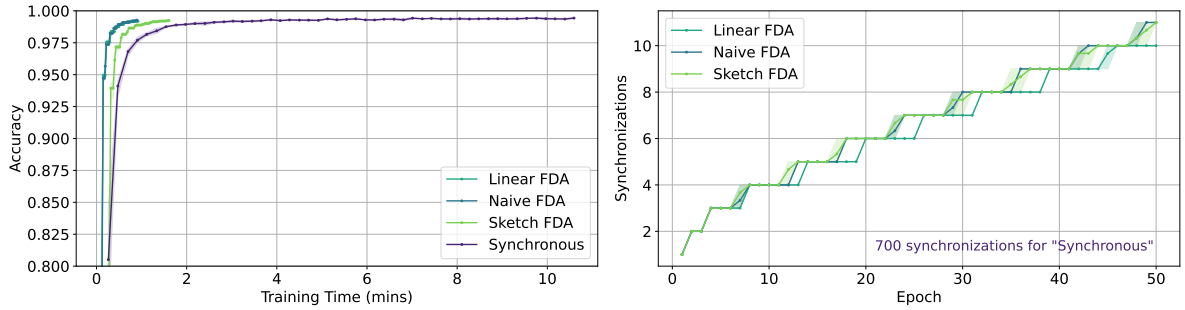


(c) Batch Size 256

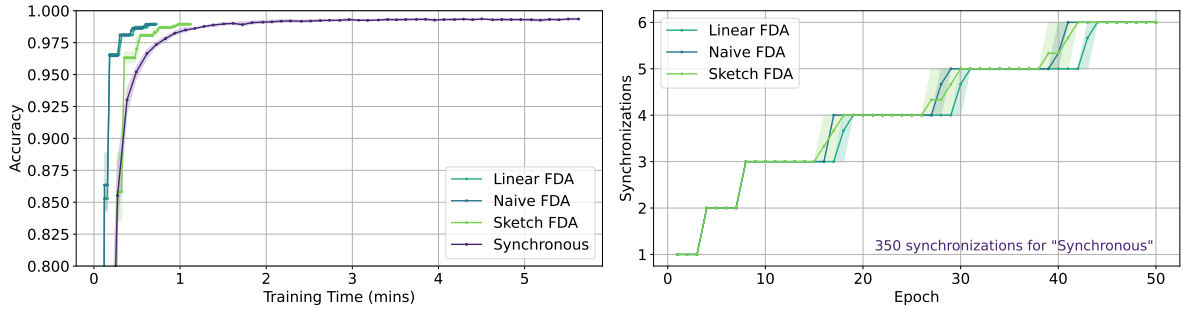
Figure A.2: Accuracy and synchronizations metrics with variable batch size (8 Clients)



(a) Batch Size 64

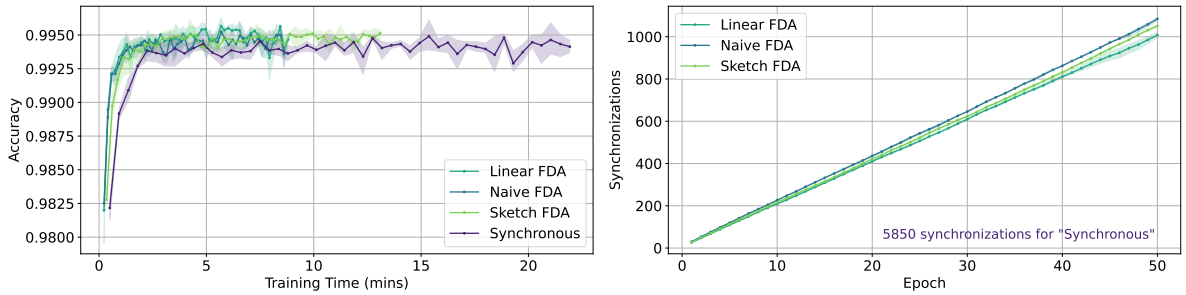


(b) Batch Size 128

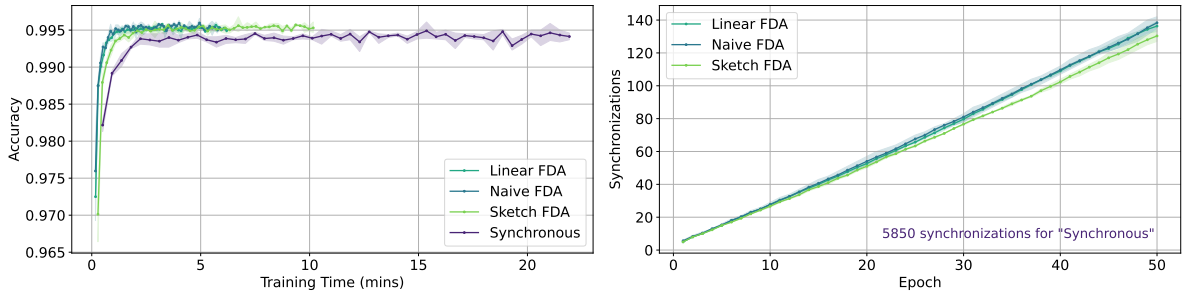


(c) Batch Size 256

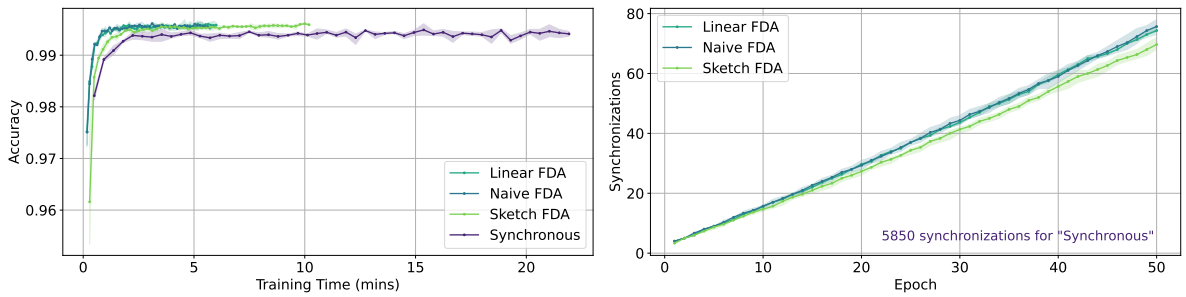
Figure A.3: Accuracy and synchronizations metrics with variable batch size (32 Clients)



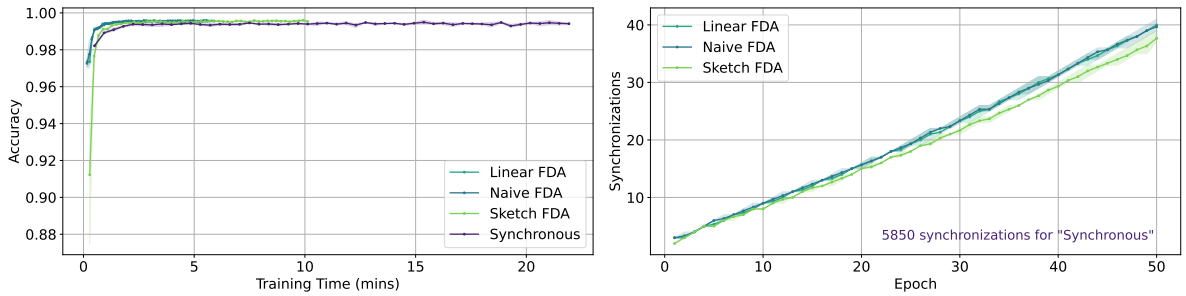
(a) Threshold 1



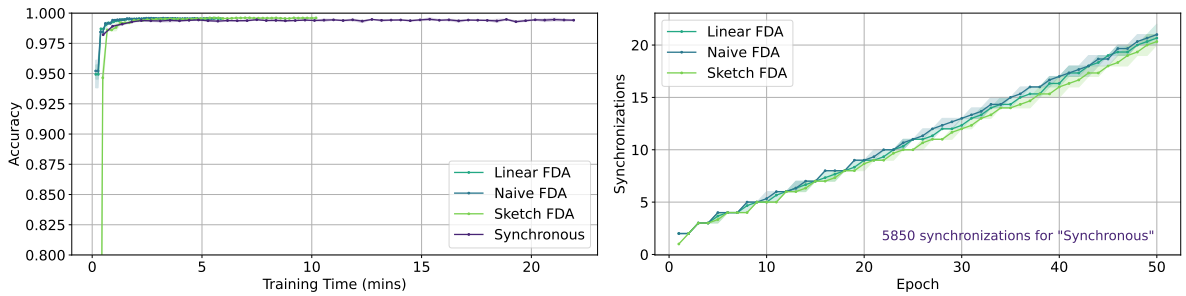
(b) Threshold 25



(c) Threshold 50

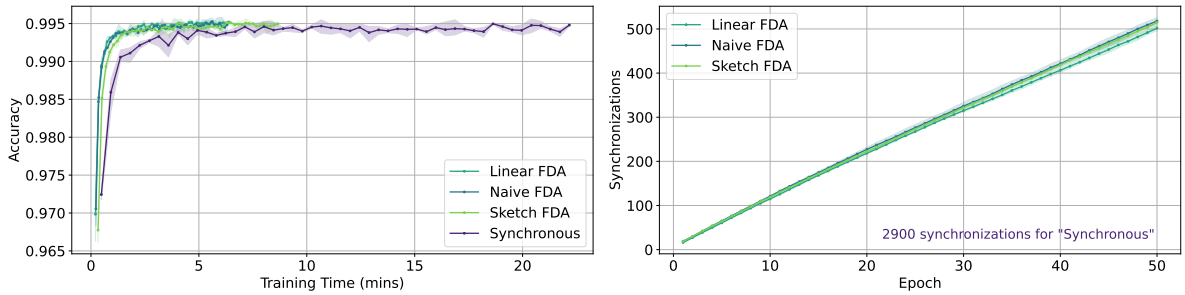


(d) Threshold 100

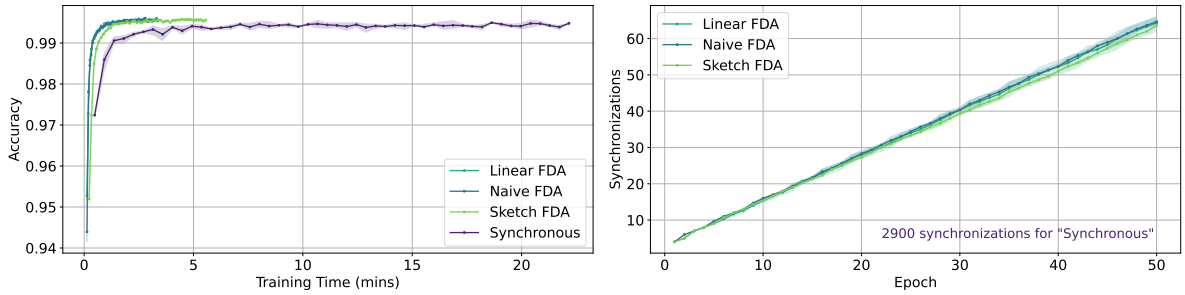


(e) Threshold 200

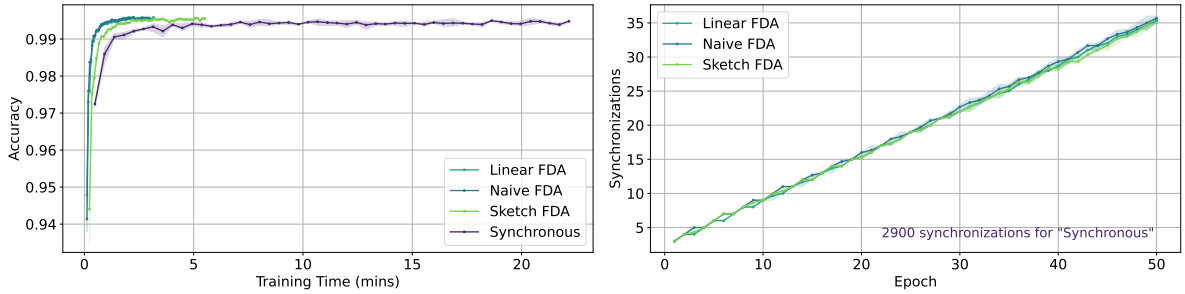
Figure A.4: Accuracy and synchronizations metrics with variable threshold (4 Clients)



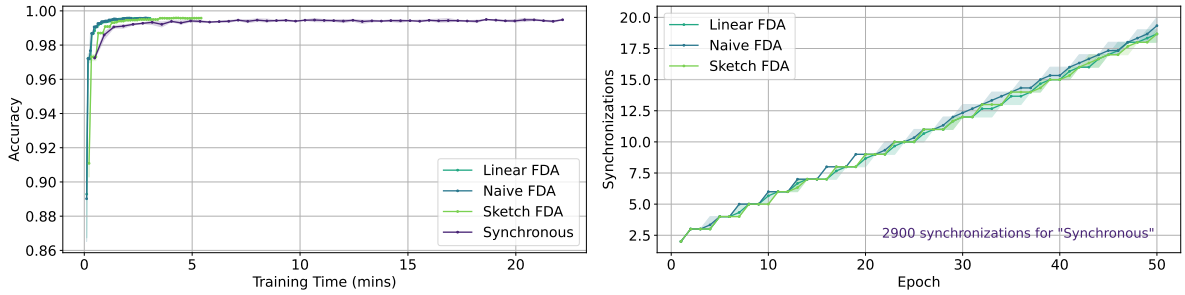
(a) Threshold 1



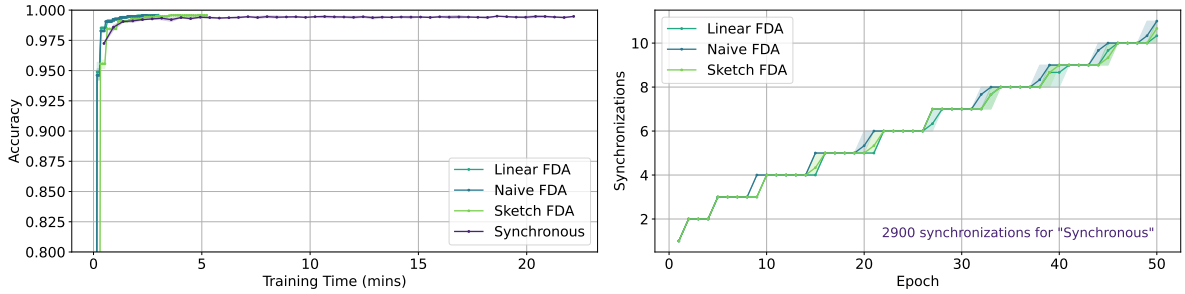
(b) Threshold 25



(c) Threshold 50

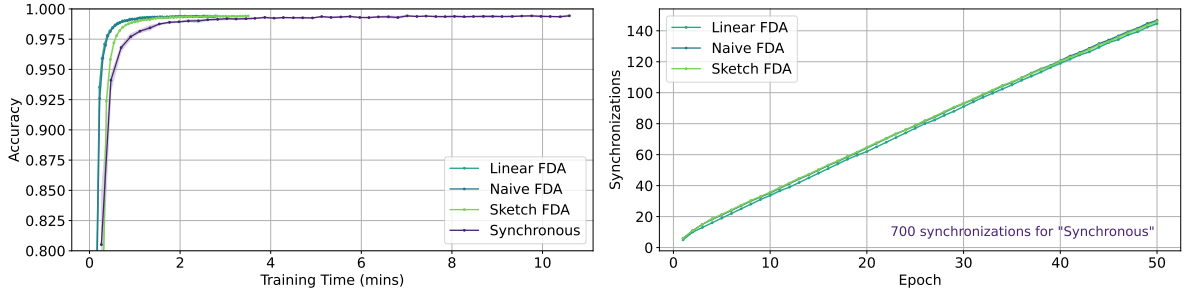


(d) Threshold 100

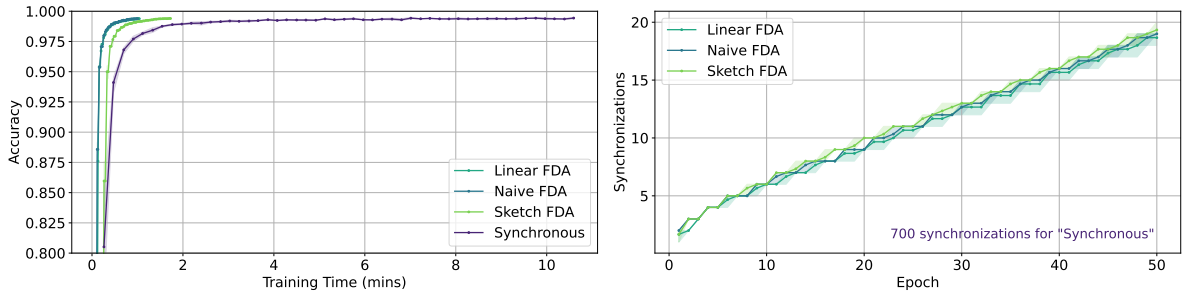


(e) Threshold 200

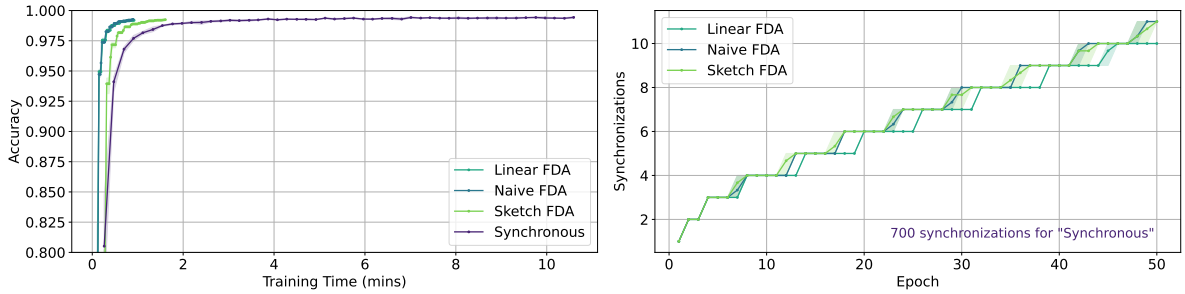
Figure A.5: Accuracy and synchronizations metrics with variable threshold (8 Clients)



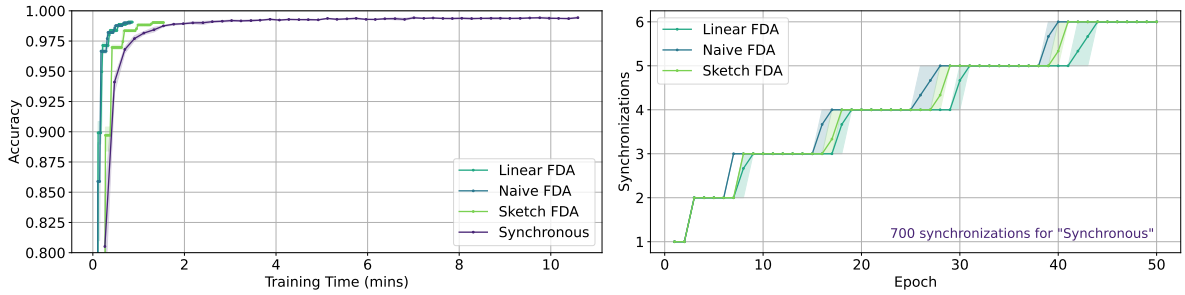
(a) Threshold 1



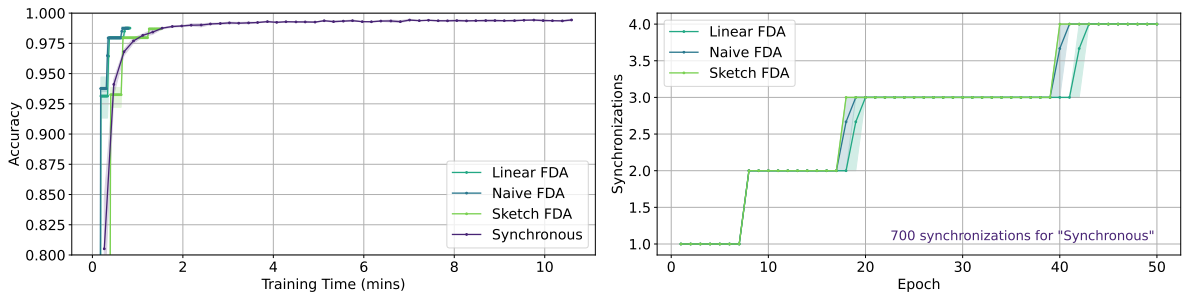
(b) Threshold 25



(c) Threshold 50



(d) Threshold 100



(e) Threshold 200

Figure A.6: Accuracy and synchronizations metrics with variable threshold (32 Clients)

Clients	Synchronous	Naive FDA	Linear FDA	Sketch FDA
4	21.93	5.72	6	10.2
8	22.17	3.03	3.16	5.51
16	16.62	1.63	1.7	2.93
32	10.59	0.9	0.92	1.61

Table A.1: Mean experiment duration (minutes) for variable number of clients

Batch Size	Synchronous	Naive FDA	Linear FDA	Sketch FDA
64	32.31	2.54	2.94	5.16
128	16.62	1.63	1.7	2.93
256	8.68	1.27	1.32	1.96

Table A.2: Mean experiment duration (minutes) for variable batch size

Threshold	Synchronous	Naive FDA	Linear FDA	Sketch FDA
1	16.62	4.3	4.35	5.59
25	16.62	1.8	1.86	3.11
50	16.62	1.63	1.7	2.93
100	16.62	1.54	1.61	2.82
200	16.62	1.49	1.57	2.8

Table A.3: Mean experiment duration (minutes) for variable threshold

Bibliography

[1] Shun-ichi Amari. "Backpropagation and stochastic gradient descent method". In: *Neurocomputing* 5.4 (1993), pp. 185–196. ISSN: 0925-2312. DOI: https://doi.org/10.1016/0925-2312(93)90006-O. URL: https://www.sciencedirect.com/science/article/pii/092523129390006O.

[2] *Anaconda Software Distribution*. Version Vers. 2-2.4.0. 2020. URL: https://docs.anaconda.com/.

[3] *ARIS HPC*. 2016. URL: https://www.hpc.grnet.gr/en/.

[4] Léon Bottou. "Stochastic Gradient Learning in Neural Networks". In: 1991. URL: https://api.semanticscholar.org/CorpusID:12410481.

[5] Nader Bouacida and Prasant Mohapatra. "Vulnerabilities in Federated Learning". In: *IEEE Access* 9 (2021), pp. 63229–63249. DOI: 10.1109/ACCESS.2021.3075203.

[6] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Mar. 2004. ISBN: 0521833787. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike-20%5C&path=ASIN/0521833787.

[7] Andrei-Octavian Brabete, Peter Pietzuch, and Daphné Tuncer. *Kungfu: A Novel Distributed Training System for TensorFlow using Flexible Synchronisation*. 2019. URL: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1819-ug-projects/BrabeteA-Kungfu-A-Novel-Distributed-Training-System-for-TensorFlow-using-Flexible-Synchronisation.pdf.

[8] François Chollet. *keras*. https://github.com/fchollet/keras. 2015.

[9] Graham Cormode and Minos Garofalakis. "Sketching Streams through the Net: Distributed Approximate Query Tracking". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 13–24. ISBN: 1595931546.

[10] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale". In: *Communications of the ACM* 56 (2013), pp. 74–80. URL: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[11] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[12] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[13] "Federated Learning Ekkono Solutions AB". In: 2020.

[14] Alex Galakatos, Andrew Crotty, and Tim Kraska. "Distributed Machine Learning". In: Jan. 2017, pp. 1–6. DOI: 10.1007/978-1-4899-7993-3_80647-1.

[15] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. "Sketch-Based Geometric Monitoring of Distributed Stream Queries". In: *Proc. VLDB Endow.* 6.10 (Aug. 2013), pp. 937–948. ISSN: 2150-8097. DOI: 10.14778/2536206.2536220. URL: https://doi.org/10.14778/2536206.2536220.

[16] Priya Goyal et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *CoRR* abs/1706.02677 (2017). arXiv: 1706.02677. URL: http://arxiv.org/abs/1706.02677.

[17] Sylvain Jeaugey. *Distributed Training and Fast inter-GPU Communication with NCCL*. GTC Silicon Valley. Presentation. Nvidia, 2019. URL: https://www.nvidia.com/en-us/on-demand/session/gtcsiliconvalley2019-s9656/.

[18] Georgios Karystinos and Vassilis Diakoloukas. *Artificial Neural Networks*. Statistical Modeling and Pattern Recognition. Technical University of Crete, Chania, Greece. 2023.

[19] Robert Kennedy. "A parallel and distributed stochastic gradient descent implementation using commodity clusters". American English. In: *Journal of Big Data* 6.1 (Dec. 2019). DOI: 10.1186/s40537-019-0179-2.

[20] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[21] Claude Lemar´echal. "Cauchy and the Gradient Method". In: (2010). URL: https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf.

[22] Li Li et al. "A review of applications in federated learning". In: *Computers & Industrial Engineering* 149 (2020), p. 106854. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2020.106854. URL: https://www.sciencedirect.com/science/article/pii/S0360835220305532.

[23] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[24] Luo Mai et al. "KungFu: Making Training in Distributed Machine Learning Adaptive". In: (2020). URL: https://www.usenix.org/system/files/osdi20-mai.pdf.

[25] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[26] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[27] H. Brendan McMahan et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. 2023. arXiv: 1602.05629 [cs.LG].

[28] Forum MPI. *MPI: A Message-Passing Interface*. Tech. rep. 1994.

[29] Feng Niu et al. *HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent*. 2011. arXiv: 1106.5730 [math.OC].

[30] Adam Paszke et al. "Automatic differentiation in PyTorch". In: 2017. URL: https://api.semanticscholar.org/CorpusID:40027675.

[31] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[32] Vasilis Samoladas and Minos N. Garofalakis. "Functional Geometric Monitoring for Distributed Streams". In: *International Conference on Extending Database Technology*. 2019. URL: https://api.semanticscholar.org/CorpusID:81985918.

[33] Vasilis Samoladas and Vissarion Konidaris. "Extreme-Scale Online Machine Learning On Stream Processing Platforms". In: *Unpublished Manuscript* (2023).

[34] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *CoRR* abs/1802.05799 (2018). arXiv: 1802.05799. URL: http://arxiv.org/abs/1802.05799.

[35] Izchak Sharfman, Assaf Schuster, and Daniel Keren. "A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams". In: *Ubiquitous Knowledge Discovery: Challenges, Techniques, Applications*. Ed. by Michael May and Lorenza Saitta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 163–186. ISBN: 978-3-642-16392-0. DOI: 10.1007/978-3-642-16392-0_10. URL: https://doi.org/10.1007/978-3-642-16392-0_10.

[36] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[37] Paul Vanhaesebrouck, Aurélien Bellet, and Marc Tommasi. "Decentralized Collaborative Learning of Personalized Models over Networks". In: *CoRR* abs/1610.05202 (2016). arXiv: 1610.05202. URL: http://arxiv.org/abs/1610.05202.

[38] Cliff Woolley. *NCCL: Accelerated Multi-GPU Collective Communications*. SC15 Conference. Presentation. Nvidia, 2015. URL: https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf.

[39] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[40] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.

[41] Sixin Zhang, Anna Choromanska, and Yann LeCun. *Deep learning with Elastic Averaging SGD*. 2015. arXiv: 1412.6651 [cs.LG].

[42] Hangyu Zhu et al. "Federated learning on non-IID data: A survey". In: *Neurocomputing* 465 (2021), pp. 371–390. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.07.098. URL: https://www.sciencedirect.com/science/article/pii/S0925231221013254.