# Algorithms for Online Federated Machine Learning

**Author:** Michail Theologitis

**Supervisor:** Vasilis Samoladas

A thesis submitted in fulfillment of the requirements
for the degree of Diploma in Electrical
and Computer Engineering

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Department of Electrical and Computer Engineering

Greece

September 2023

# Algorithms for Online Federated Machine Learning


## Abstract

In the contemporary landscape, the proliferation of data, driven by the internet, social media, and the Internet of Things (IoT), has catalyzed transformative shifts across various sectors, with Machine Learning (ML) standing at the forefront of this revolution. As the Big Data era fosters remarkable ML advancements in domains like image recognition and natural language processing, it also introduces substantial computational challenges, logistical hurdles, and privacy concerns. These complexities have pushed traditional ML approaches to the limit, often proving inadequate, prompting the rise of innovative paradigms such as Federated Learning (FL). We delve into the challenges posed by FL's traditional iterative training process, which is non-dynamic and prescribes predetermined operations for participating learners. We investigate a dynamic approach anchored in the principles of Functional Geometric Monitoring (FGM), a state-of-the-art technique for monitoring distributed data streams, with the aim to enhance the training process by significantly mitigating communication overhead. Through the lens of FGM and employing three approximation techniques, our work evaluates its efficacy in refining FL's conventional training procedures, supported by comprehensive experimental analyses.

# Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Vasilis Samoladas. His unwavering support, meticulous guidance, and willingness to always find time to help have been the cornerstone of this thesis. The many hours he devoted to explaining concepts, providing feedback, and nurturing my understanding have been invaluable. Furthermore, his courses on "Computational Geometry" and "Principles of Distributed Systems" not only provided me with invaluable knowledge but also led to a paradigm shift in my thinking, unparalleled by any other experience in my academic journey. Through his persistent interest and encouragement, I have not only gained profound knowledge in the field but have also been deeply inspired. The impact of his mentorship, both in this work and in my broader academic and personal development, extends far beyond this thesis.

I am also immensely thankful to Professor Athanasios Liavas. His course on "Optimization" equipped me with the foundational knowledge essential for delving into and understanding complex Machine Learning literature. Beyond the lectures, his keen interest in my progress and valuable recommendations have been a consistent source of guidance.

My appreciation extends to Professor Antonios Deligiannakis for his interest in my work. Our bi-weekly meetings, alongside other students and faculty, provided a platform for engaging discussions on Federated Learning and related areas. These interactions have been instrumental in broadening my perspective and refining my approach.

I also wish to express my gratitude to Professor Minos Garofalakis for his role in the committee and for his time and consideration regarding my work.

Lastly, I cannot emphasize enough the invaluable support from my family. My parents and brother, in particular, have continuously buoyed my spirits with their unwavering belief in my potential.

# Contents

# Chapter 1

# Introduction

## 1.1 Machine Learning in the Big Data Era

In the modern era, data has become an invaluable resource. The advent of the internet, social media, and the Internet of Things (IoT) has led to an explosion in the volume, velocity, and variety of data being generated every second. This phenomenon, known as Big Data, has transformed numerous fields, from business and healthcare to science and engineering. One field that has particularly benefited from this data deluge is Machine Learning (ML).

The surge in available data has provided ample opportunities for ML algorithms to evolve and improve, leading to remarkable advancements in areas such as image recognition, natural language processing, and predictive analytics.

However, the Big Data era is not without its challenges. The sheer volume of data and the computational resources required to process it are becoming increasingly demanding. Traditional approaches may struggle to keep up, leading to inefficiencies and potential privacy concerns. These challenges have spurred innovation and the development of new paradigms, such as Distributed Machine Learning and Federated Learning, which we will explore throughout this thesis.

## 1.2 Distributed Machine Learning & Federated Learning

### 1.2.1 Overview

Have you ever pondered the process behind training intricate models that encompass millions or even billions of parameters using data that spans terabytes? It's mind-boggling to consider that some of these models are so massive that they surpass the memory limits of a single processor. Traditional training methods fall short under such burdens, prompting the need for innovative techniques. One standout solution is *Distributed learning*. Central to the machine learning frameworks of today's leading tech companies, it capitalizes on the combined processing power of numerous machines. By doing so, it facilitates the swift training of extensive models on vast datasets, enabling the creation of top-notch models that can be refined at an accelerated rate. The following is Google's take on the matter more than a decade ago [1] :

*"Our experiments show that the distributed optimization approach can both greatly accelerate the training of modestly sized models, and that it can also train models that are larger than could be contemplated otherwise."*

Generally, when faced with a gigantic task in any field, we try to divide it into subtasks and run them in parallel. This saves us time and makes such a complex task doable. When we do the same in deep learning, we call it *Distributed Learning*. Distributed ML

algorithms have received considerable attention over the last years as data continuously grows and the models become more and more complex. Even though we try to refrain from diving into technical details at this point, we have to note the following: most Distributed ML algorithms entail some sort of data partitioning[1] (splitting the data and assigning parts to individual learners for parallelization) - this fact is intuitive, after all. However, the underlying assumption is that we have access to the entire data-set during training. Furthermore, our main objective is to speed-up and make possible the training process. But what if the data can not be centralized? What if the data is confidential (e.g. medical records [2]) and malicious adversaries come into play? What if we want to train a model under such challenging conditions?

These are some of the questions and concerns that gave birth to *Federated Learning* (FL). It was first introduced as a term in 2016 by Google [3]. During this period, there was escalating awareness around the world about the potential pitfalls of personal data handling. Notably, events like the Cambridge Analytica incident heightened user caution on platforms like Facebook, spotlighting the risks associated with sharing personal data. In contrast to Distributed Learning which aims at parallelizing computing power, Federated Learning aims at training on heterogeneous data-sets in settings where data centralization is infeasible, adversaries are not ruled out, and privacy is one of the top concerns [4]. Their main difference lies in the assumptions made on the properties of the local data-sets. The need for a comprehensive survey paper addressing the numerous open challenges in the field of FL was underscored during the Workshop on Federated Learning and Analytics, held on June 17-18, 2019, at Google's Seattle office. Consequently, a seminal paper was developed [4], which provided the following definition of Federated Learning:

*"Federated learning is a machine learning setting where multiple entities (clients) collaborate in solving a machine learning problem, under the coordination of a central server or service provider. Each client's raw data is stored locally and not exchanged or transferred; instead, focused updates intended for immediate aggregation are used to achieve the learning objective."*

One of the first deployments of FL was Google's Gboard in 2017. The project aimed to be "privacy forward" by implementing strong guidelines for identifying data that should be considered classified. Rather than centralizing this data, Gboard utilized an on-device cache of local interactions. This approach showcased the potential of FL in a real-world application.

### 1.2.2 Learning Process

In both Distributed Machine Learning and Federated Learning, the learning process involves multiple clients (e.g. devices or organizations) that collaboratively aim to train a global model. The process can be broken down into *rounds*, with each *round* consisting of two main steps: *local training* and *model aggregation*.

During the local training step, each client independently trains the model on its own local data. This is typically done using a method like Stochastic Gradient Descent (SGD) and variants.

Once clients have completed their local training, the model aggregation begins. In this step, a central server [2] collects the locally trained models from each client and combines them into a single, global model. This is often done by averaging the collected models.

The global model is then sent back to the clients, and the next round of local training begins. This process continues until the model's performance on a validation set stops

[1] This statement is a simplification. Some distributed learning approaches, such as model-parallelism, may not involve data partitioning.

[2] Decentralized approaches are also common, but in this introduction, we focus on the centralized scenario to convey the key concepts.

improving or after a predetermined number of *rounds* (or *epochs*) is reached.

The main difference between Distributed Machine Learning and Federated Learning lies in where the data is stored and how it's used. In Distributed Machine Learning, the data can be centrally stored and partitioned among the clients for parallel processing. In contrast, in Federated Learning, the data remains on the clients' devices, and only the model parameters are exchanged, usually under the promise of *Differential Privacy* (which guarantees data privacy) [4].

In the context of this thesis, the model aggregation step, also known as *synchronization*, is of particular interest. The synchronization process can be resource-intensive, especially in terms of communication cost, due to the potentially large number of clients and the size of the models involved. This sets the stage for the problem that this thesis aims to address, which will be discussed in more detail in Section 1.4.

Bear in mind that this is a high-level overview of the learning process in Distributed Machine Learning and Federated Learning. We'll delve into more technical details later on.

## 1.3 Monitoring Non-linear Functions on Distributed Data Streams

In the era of Big Data, the information infrastructure of modern societies is increasingly defined by data flowing continuously from diverse sources. This data influx poses challenges for data centralization given the immense communication expenses associated with consolidating all data into one processing hub. These challenges have spurred extensive research focused on monitoring complex ongoing queries across vast and swiftly moving distributed streams. Two methods that have emerged from this research are Geometric Monitoring (GM) and Functional Geometric Monitoring (FGM). *Geometric Monitoring* (GM), introduced in [5, 6], utilizes geometric principles to reduce the communication overhead in monitoring distributed data streams by representing data streams in a high-dimensional space and defining boundaries. *Functional Geometric Monitoring* (FGM), which can be seen as a generalization of GM, was introduced by Vasilis Samoladas and Minos Garofalakis [7]. FGM's approach involves applying specific non-linear functions to transform the multi-dimensional data points, allowing for more efficient and adaptable monitoring of distributed data streams [8, 9, 7].

While GM and FGM were initially developed for monitoring distributed data streams, their principles and techniques have potential applications in Distributed Machine Learning (ML) and Federated Learning (FL). In the journey ahead, we will explore how these methods can be adapted to alleviate communication challenges in FL, providing innovative solutions for modern data-driven challenges.

## 1.4 Objectives & Methodology

The primary objective of this thesis is to explore algorithms and techniques specifically designed for Federated Learning (FL), with a particular focus on the synchronization process. This process, which involves the aggregation of client models by a central coordinator, is resource-intensive and presents a significant challenge to the scalability of FL. Specifically, the communication overhead incurred by clients sending their models over the network for aggregation is a major bottleneck in FL. This is due to the potentially large size of the models and the number of clients involved, which can lead to substantial network congestion and latency. Therefore, techniques that can reduce this communication overhead are of paramount importance for improving the efficiency and scalability of FL.

In the Federated Learning (FL) setting, training typically follows a round-based approach, where each round is carefully structured and predefined, complete with specific local steps and local epochs (details of which we will explore later on). This fixed structure, dictating when the round terminates and synchronization occurs, has become the standard. However, we aim to investigate an alternative approach that leverages the principles of Geometric Monitoring (GM) and its generalization, Functional Geometric Monitoring (FGM). This approach seeks to dynamically determine when rounds should be terminated, aiming to minimize unnecessary synchronizations and reduce communication overhead. In this work, we will investigate this innovative method, delving into its potential to reshape the conventional training process in FL.

While the Functional Geometric Monitoring (FGM) protocol provides a robust theoretical foundation, our experimental work will primarily leverage three simplified approximation methods: the "naive", "linear", and "sketch" approaches. Although we may not strictly adhere to the standard FGM protocol in our experiments, the principles and insights derived from FGM will guide our methodology and analysis throughout this work. The specific measure we monitor, along with its significance, will be introduced and explored in detail later on.

We will apply this approach to training two different Convolutional Neural Networks: a relatively small network and a more complex, deeper network. Our goal is to determine the extent of improvement offered by the FGM approach in the Federated Learning (FL) setting. To achieve this, we will conduct a thorough experimental evaluation, including a comprehensive exploration of hyper-parameters. This rigorous methodology will ensure the robustness of our findings and provide a detailed understanding of the FGM approach's performance and potential applications in FL.

## 1.5 Outline

- **Chapter 2: Related Work** - This chapter lays the groundwork for the thesis by introducing fundamental concepts in Machine Learning, descent methods, Distributed Machine Learning, and Artificial Neural Networks. Additionally, the chapter offers an overview of the TensorFlow platform and its high-level API, Keras. It builds the necessary knowledge to understand the Federated Learning (FL) setting and training in FL, ensuring a smooth transition to the more advanced topics in the subsequent chapters.

- **Chapter 3: Functional Dynamic Averaging** - This chapter delves into the Functional Dynamic Averaging (FDA) strategy, adapted to suit the Federated Learning setting, with the primary goal of minimizing communication overhead during training. The concept of weighted variance is introduced that serves as a gauge, indicating if training is progressing in the desired direction, forming the foundation for the Round Terminating Condition (RTC). The chapter then explores three distinct approximation methods to monitor the RTC: the "naive," "linear," and "sketch" approaches.

- **Chapter 4: Experiments** - The chapter delves into the empircal evaluation of the Functional Dynamic Averaging (FDA) algorithm within the Federated Learning (FL) setting. We compare the three FDA strategies ("naive," "linear," and "sketch") with the baseline "synchronous" strategy. A significant portion is dedicated to detailing our custom-developed Python package, a simulator tailored for the FDA algorithm in the FL setting.

- **Chapter 5: Conclusion** - This chapter concludes the thesis with a summary of our work and outlines potential directions for future exploration.

# Chapter 2

# Related Work

## 2.1 Machine Learning (ML)

Machine Learning (ML) is a subset of artificial intelligence centered around crafting algorithms and statistical frameworks that empower computers to undertake particular tasks, all without relying on explicit instructions. Essentially, it's about equipping machines with the ability to assimilate data, identify patterns, and execute decisions with minimal human intervention. Machine learning can be broadly categorized into two main types: Unsupervised Learning and Supervised Learning. In this thesis, we will explore various aspects of machine learning, with a particular emphasis on supervised learning, as it plays a central role in many modern applications and forms the basis for the methods and concepts discussed in the following sections.

### 2.1.1 Supervised Learning

Supervised learning is a category of machine learning where the model is trained on a labeled dataset. This means that for every input in the dataset, there is a corresponding known output. The model's task is to learn the underlying relationship between the inputs and outputs, allowing it to make predictions on unseen data.

In a typical supervised learning problem, we have a dataset consisting of *examples* or *samples*, where each example is made up of a *feature vector* and a corresponding *label* or *class*. The feature vector represents the input, and the label represents the desired output. The dataset can be denoted as $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where each $x_i$ represents a feature vector, and each $y_i$ represents the corresponding label.

Supervised learning encompasses a wide variety of tasks, including regression, where the output is a continuous value, and classification, where the output is a discrete class label. Common methods used in supervised learning include linear regression, logistic regression, support vector machines, and the application of various types of neural networks.

The training process typically involves iteratively adjusting the model's parameters to minimize a loss function (objective function), which quantifies the discrepancy between the model's predictions and the actual target values in the training data. This process is guided by various optimization techniques, including Stochastic Gradient Descent (Section 2.2.2), among others that will be explored in the following sections.

### 2.1.2 Unsupervised Learning

Unsupervised learning is another major category of machine learning that operates without labeled outputs in the training data. Unlike supervised learning, where the goal is to learn a mapping from inputs to known outputs, unsupervised learning seeks to discover hidden patterns, structures, or features within the data itself.

The primary tasks in unsupervised learning include clustering, dimensionality reduction, and anomaly detection. Clustering aims to group similar examples together based on some measure of similarity, such as Euclidean distance. Dimensionality reduction seeks to represent the data in a lower-dimensional space, preserving as much of the relevant information as possible. Anomaly detection focuses on identifying abnormal or unusual patterns that differ significantly from the majority of the data. Common algorithms and techniques used in unsupervised learning include k-means clustering, hierarchical clustering, Principal Component Analysis (PCA), and auto-encoders.

Unsupervised learning offers unique advantages, particularly when labeled data are scarce or expensive to obtain. It can be used to explore the underlying structure of the data, identify hidden relationships, and provide insights that may not be apparent through supervised methods.

Applications of unsupervised learning are diverse and include customer segmentation in marketing, topic modeling in text analysis, image compression, and fraud detection in financial transactions.

## 2.2 Descend Methods

Consider a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$. We will refer to $f$ as the *objective function*. We want to solve the unconstrained optimization problem:

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}} \;\; f(w)$$

The algorithms described in this section produce a minimizing sequence $w_t$, $t = 1, \ldots,$ where

$$w_{t+1} = w_t + m_t \Delta w_t$$

and $m_t > 0$ (except when $w_t$ is optimal). Here the concatenated symbols $\Delta$ and $w$ that form $\Delta w_t$ are to be read as a single entity, a vector in $\mathbb{R}^d$ called the *step* or *search direction*, and $t = 1, \ldots,$ denotes the iteration number. The scalar $m_t$ is called the *step size* or *step length* at iteration $t$. We will drop the superscripts and use the lighter notation $w \leftarrow w + m\Delta w$, in place of $w_{t+1} = w_t + m_t \Delta w_t$.

The outline of a general descent method is as follows: It alternates between two steps: determining a descend direction $\Delta w$, and the selection of a step size $m$.

**Algorithm 1** General descent method [10]

**given** A starting point $w \in$ **dom** $f$
**repeat**
1: Determine a descent direction $\Delta w$
2: *Line search.* Choose a step size $m > 0$
3: *Update.* $w \leftarrow w + m\Delta w$
**until** stopping criterion is satisfied

The second step is called the *line search* since selection of the step size $t$ determines where along the line (more accurately, *ray*) $\{w + m\Delta w \mid m \in \mathbb{R}_+\}$ the next iterate will be. There are many different line search methods, each with its own advantages and specific use cases [10].

### 2.2.1 Gradient Descend

A natural choice for the search direction is the negative gradient $\Delta w = -\nabla f(w)$. The resulting algorithm is called *Gradient algorithm* or *Gradient Descend method* [10].

In the context of machine learning, particularly when training deep neural networks, it's common to use a fixed *step size* (*learning rate*) denoted as $\eta$ for the gradient descent method. This is primarily due to computational inefficiency, as performing a line search at each step can be computationally expensive, especially when dealing with large datasets and complex models. The update rule becomes:

$$w \leftarrow w - \eta \nabla f(w)$$

### 2.2.2 Stochastic Gradient Descent (SGD)

*Stochastic Gradient Descent* (SGD) is a variation of the gradient descent method that computes the gradient using a single randomly chosen example from the dataset at each step, rather than the entire dataset. This makes SGD faster and more scalable to large datasets compared to the standard gradient descent method.

In the context of training a machine learning model, we typically have a dataset of examples and a loss function that measures how well the model fits each example. The goal is to find the model parameters (denoted by $w$) that minimize the average loss over all examples in the dataset. This is equivalent to minimizing the function

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w)$$

where $n$ is the number of examples in the dataset, and $f_i(w)$ is the loss for the $i$-th example. More specifically, $f_i(w)$ is the loss $l(x_i, y_i; w)$ on example $(x_i, y_i)$ with parameters $w$.

For the sake of succinctness and ease of understanding, we will use the notation $f_i(w)$ to denote the loss for the $i$-th example throughout this thesis. This notation will allow us to express complex mathematical concepts more clearly and concisely.

The update rule for SGD is similar to that of gradient descent, but with the gradient $\nabla f(w)$ replaced by a stochastic estimate of the gradient

$$w \leftarrow w - \eta \nabla f_i(w)$$

where $i$ is chosen uniformly at random from $\{1, \ldots, n\}$ at each step [11].

Despite its simplicity, SGD has proven to be highly effective for training a wide range of machine learning models, including deep neural networks. However, one of the challenges with SGD is the selection of the learning rate $\eta$. If $\eta$ is too large, SGD may fail to converge; if $\eta$ is too small, the convergence may be too slow. Various strategies have been proposed to adaptively adjust the learning rate during the course of optimization, leading to several variants of SGD.

### 2.2.3 Mini-batch Gradient Descent

One popular variant of SGD is *Mini-Batch Gradient Descent*, which computes in parallel the gradient using a small batch of examples at each step, rather than a single example. This can lead to a more stable and accurate estimate of the gradient, while still being much faster than computing the gradient over the entire dataset.

The update rule for Mini-Batch Gradient Descent is

$$w \leftarrow w - \eta \nabla f_B(w)$$

where $B$ is a mini-batch of examples chosen at random at each step. The function $f_B(w)$ is the average loss over the examples in the mini-batch, and its gradient is computed as

$$\nabla f_B(w) = \frac{1}{|B|} \sum_{i \in B} \nabla f_i(w) \tag{2.1}$$

where $|B|$ is the number of examples in the mini-batch.

Mini-batch gradient descent strikes a balance between the computational efficiency of stochastic gradient descent and the stability and accuracy of full-batch gradient descent[1]. By averaging the gradients over a mini-batch of examples, it reduces the variance of the gradient estimates, which can lead to more stable and consistent progress towards a minimum. This can be particularly beneficial in the presence of noisy data or non-smooth optimization landscapes, where the gradient can vary significantly from one example to another.

Moreover, Mini-Batch Gradient Descent is highly amenable to parallelization, as the gradients for different examples in the mini-batch can be computed simultaneously. This makes it a popular choice for training deep neural networks on modern hardware accelerators such as GPUs, which can perform many computations in parallel.

However, similar to SGD, the choice of the learning rate and the mini-batch size can significantly affect the performance of Mini-Batch Gradient Descent. A learning rate that is too large can cause the method to diverge, while a learning rate that is too small can result in slow convergence. Similarly, a mini-batch size that is too large can lead to less frequent updates, which might slow down the convergence. On the other hand, a mini-batch size that is too small can result in noisy gradient estimates. Therefore, these hyperparameters often need to be carefully tuned for each specific problem.

In the machine learning community, the term Stochastic Gradient Descent (SGD) often implicitly encompasses Mini-Batch Gradient Descent. While the original SGD algorithm computes the gradient using a single example at each step, it's common in practice to use a mini-batch of examples for computational efficiency and stability. This mini-batch variant is technically a different algorithm, but it's often simply referred to as SGD, without explicitly mentioning the use of mini-batches. The transition from single-sample SGD to its mini-batch counterpart is straightforward, which further blurs the distinction between the two in practical applications. Therefore, when interpreting machine learning literature or using machine learning software, it's important to understand that SGD might actually be implemented as Mini-Batch Gradient Descent. This understanding simplifies the discourse, as we can refer to both single-sample and mini-batch variants under the umbrella term of SGD, without the need for constant differentiation.

### 2.2.4 Adaptive Moment Estimation (Adam)

*Adaptive Moment Estimation* (Adam), introduced in 2014 [12], combines the advantages of two popular gradient descent methods: Momentum [13] and RMSProp [2]. From Momentum, Adam borrows the idea of adding fractions of previous gradients to current ones, which amplifies the speed of descent in consistent directions and dampens oscillations. From RMSProp, Adam takes the concept of normalizing the gradient by an exponential moving average of its past squared values, which helps in navigating through ravines, i.e., areas where the surface curves much more steeply in one dimension than in another.

Adam's update rule is more complex than that of SGD or Mini-Batch Gradient Descent, and it involves maintaining an exponentially decaying average of past gradients and

[1] Full-batch gradient descent refers to the variant of gradient descent where the gradient is computed over the entire dataset at each step. This method provides the most accurate estimate of the gradient, but it is computationally expensive and less scalable.

[2] Unusually, RMSProp was not published in an article but merely described in a Coursera lecture.

past squared gradients. This is why we will present it in the form of an algorithm, to provide a complete and clear picture of all the steps involved.

**Algorithm 2** Adam: Adaptive Moment Estimation [12]

**Require:** $\eta$: Step size (learning rate)
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(w)$: Stochastic objective function with parameters $w$
**Require:** $w_0$: Initial parameter vector
$m_0 \leftarrow 0$ ▷ Initialize 1st moment vector
$v_0 \leftarrow 0$ ▷ Initialize 2nd moment vector
$t \leftarrow 0$ ▷ Initialize step
**repeat**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_w f_t(w_{t-1})$ ▷ Get gradients w.r.t. stochastic objective at step $t$
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Update biased first moment estimate
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ ▷ Update biased second raw moment estimate
$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected first moment estimate
$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected second raw moment estimate
$w_t \leftarrow w_{t-1} - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ ▷ Update parameters
**until** $w_t$ has converged
**return** $w_t$ ▷ Resulting parameters

In the context of the Adam algorithm (Algorithm 2), it is important to clarify certain elements. The term $\epsilon$ represents a small scalar that serves the purpose of preventing division by zero. All operations performed on vectors are executed element-wise. The notation $g_t^2$ indicates the element-wise square $g_t \odot g_t$. Finally, the expressions $\beta_1^t$ and $\beta_2^t$ denote $\beta_1$ and $\beta_2$ raised to the power $t$.

Adam has been widely adopted in the machine learning community due to its efficiency and robustness. It is particularly suitable for problems with large data or parameters, such as deep learning, and can handle sparse gradients on noisy problems, which are common in natural language processing and computer vision tasks.

One of the key features of Adam is its adaptive learning rate mechanism. While the global learning rate $\eta$ is fixed, the effective learning rate for each parameter is adaptive. This is achieved by the update rule in Adam, which divides the first moment estimate (an estimate of the mean of the gradients) by the square root of the second moment estimate (an estimate of the variance of the gradients). This scales the learning rate for each parameter inversely proportional to the square root of the sum of the squares of its past gradients. As a result, parameters with larger average past gradients have smaller effective learning rates, and vice versa. This mechanism allows the learning rate to change over time based on the historical gradients for each parameter, making Adam often described as having an "adaptive learning rate" mechanism.

Moreover, Adam is user-friendly for practitioners as it requires little tuning of hyper-parameters. The defaults usually work well, with the only parameter that may require tuning in practice being the learning rate $\eta$. The exponential decay rates for the moment estimates are usually initialized as $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and the fuzz factor $\epsilon$ is usually set to $10^{-8}$.

In conclusion, Adam is a powerful and versatile optimization algorithm that can handle a wide range of optimization problems with ease and efficiency. It has become a go-to choice for many machine learning practitioners and researchers for training deep neural networks.

## 2.3 Distributed Machine Learning (DML)

Distributed Machine Learning (DML) is a subfield of machine learning that focuses on training models across multiple computational nodes, or "nodes" for short. This approach is primarily motivated by the need to handle large-scale data and complex models that exceed the capacity of a single machine. By distributing the computational load across multiple nodes, DML enables faster training times and the ability to handle larger models and datasets.

In DML, the training process is typically organized into a series of iterations. In each iteration, every node performs computations on its local data or model parameters and communicates the results to a central entity, often referred to as the *coordinator*[3]. The coordinator then aggregates the results from all nodes to update the global model. This process continues until the model's performance meets a predefined criterion or after a certain number of iterations.

The specifics of the training process can vary depending on the particular DML algorithm used. For instance, in Synchronous Stochastic Gradient Descent (SSGD), updates from all nodes are collected before updating the global model, thus forming distinct rounds. On the other hand, in Asynchronous Stochastic Gradient Descent (ASGD), updates from the nodes are incorporated into the global model as soon as they arrive at the coordinator, without waiting for updates from all nodes.

### 2.3.1 Architecture

The architecture of a DML system is primarily composed of a coordinator and multiple nodes.

The *coordinator*, also known as the *parameter server* or *master*, orchestrates the training process. It is responsible for managing the training process, distributing the data or model parameters to the nodes, aggregating the updates from the nodes, updating the global model, and checking for convergence. The coordinator ensures that all nodes are working in harmony towards the common goal of training the model.

On the other hand, the *nodes*, also known as *workers* or *clients*, are the computational units in a DML system. Each node performs computations on its local data or model parameters and communicates the results to the coordinator. The nodes operate independently of each other, allowing for parallel computation and thus speeding up the training process.

In the following sections, we will delve into the specifics of how the training process is carried out in a DML system, focusing on two key algorithms: Asynchronous Stochastic Gradient Descent (ASGD) and Synchronous Stochastic Gradient Descent (SSGD). These algorithms leverage the DML architecture to efficiently train models on large-scale data.

### 2.3.2 Asynchronous Stochastic Gradient Descend (ASGD)

The Asynchronous Stochastic Gradient Descent (ASGD) algorithm is a variant of the Stochastic Gradient Descent (SGD) algorithm that allows for parallel and asynchronous updates of the model parameters. This makes it particularly suitable for distributed computing environments, where multiple workers can compute gradients and update the model parameters independently and simultaneously. The ASGD algorithm works as follows:

1. Each worker independently selects a data point and computes the stochastic estimate of the gradient of the loss function with respect to the model parameters.

[3]While we focus on centralized DML in this thesis, it's worth noting that there are also decentralized or peer-to-peer learning frameworks that do not rely on a central coordinator.

2. Each worker then sends its computed gradient to a parameter server, which maintains the current estimate of the model parameters.

3. The parameter server updates the model parameters asynchronously as soon as it receives a gradient from a worker. This means that the parameter server does not wait for gradients from all workers before updating the model parameters, which is what makes ASGD asynchronous.

4. Each worker fetches the latest model parameters from the parameter server before computing the next gradient.

5. This process continues until the model parameters have converged, or until a pre-specified stopping criterion is met.

The Asynchronous Stochastic Gradient Descent (ASGD) algorithm can be succinctly described by the following update rule, which is executed by the parameter server at each iteration:

$$w_{t+1} \leftarrow w_t - \eta \nabla f_{\gamma_t}(w_{t-\tau_t})$$

In the update rule, $\eta$ represents the step size or learning rate, and $\gamma_t$ is an index chosen uniformly at random from $\{1, \ldots, n\}$, representing a selected data point. The term $w_{t-\tau_t}$ denotes a delayed version of the model parameters, read by a worker, where the delay, or *staleness*, is denoted by $\tau_t$. Moreover, $\nabla f_{\gamma_t}(w_{t-\tau_t})$ is the stochastic estimate of the gradient of the loss for the $\gamma_t$-th example (data-point), computed by a worker using the delayed parameters $w_{t-\tau_t}$, and sent to the parameter server to perform the update [14].

While the description of ASGD above uses a single data point for each gradient computation, in practice, it is common to use a mini-batch of data points, similar to Mini-Batch Gradient Descent. This is often implicitly understood, and the term 'ASGD' may be used to refer to both the single data point and mini-batch versions of the algorithm. The transition from using a single data point to a mini-batch of data points is straightforward, similar to the transition from SGD to Mini-Batch Gradient Descent (as discussed in 2.2.3).

ASGD offers significant speedup in the training process in a distributed computing environment, as it allows for simultaneous computation and application of gradients. However, its asynchronous nature can introduce additional noise into the optimization process, as the model parameters can be updated with stale gradients, i.e., gradients computed with respect to an outdated version of the model parameters. This can potentially slow down the convergence of the algorithm, or even cause it to diverge. The issue of stale gradients has been addressed in several research studies. For instance, Zhu and Ying [15] provided a sharp convergence rate for ASGD when the loss function is a perturbed quadratic function, showing that longer delays result in slower convergence rate. Similarly, Zhang et al. [16] proposed a variant of the ASGD algorithm in which the learning rate is modulated according to the gradient staleness, providing theoretical guarantees for the convergence of this algorithm. Despite these challenges, ASGD remains a powerful tool for large-scale machine learning tasks.

### 2.3.3 Synchronous Stochastic Gradient Descent (SSGD)

The Synchronous Stochastic Gradient Descent (SSGD) algorithm is another variant of the Stochastic Gradient Descent (SGD) algorithm that is designed for distributed computing environments. Unlike ASGD, SSGD operates in a synchronous manner, meaning that all workers must complete their computations and send their gradients to the parameter server before the model parameters are updated. The SSGD algorithm works as follows:

1. Each worker independently selects a data point or a mini-batch of data points and computes the stochastic estimate of the gradient of the loss function with respect to the model parameters.
2. Each worker then sends its computed gradient to the parameter server, which maintains the current estimate of the model parameters.
3. The parameter server waits until it has received gradients from all workers. It then averages these gradients and updates the model parameters synchronously.
4. Each worker fetches the latest model parameters from the parameter server before computing the next gradient.
5. This process continues until the model parameters have converged, or until a prespecified stopping criterion is met.

The Synchronous Stochastic Gradient Descent (SSGD) algorithm can be succinctly described by the following update rule, which is executed by the parameter server at each iteration:

$$w_{t+1} \leftarrow w_t - \eta \frac{1}{K} \sum_{i=1}^{K} \nabla f_{\gamma_{t,i}}(w_t)$$

In the update rule, $\eta$ represents the step size or learning rate, $K$ the number of workers, and $\gamma_{t,i}$ is an index chosen uniformly at random by the i-th worker from $\{1, \ldots, n\}$, representing a selected data point. Each $\nabla f_{\gamma_{t,i}}(w_t)$ is the stochastic estimate of the gradient, from the $i$-th worker, of the loss for the $\gamma_{t,i}$-th example (data-point). The term $\frac{1}{K} \sum_{i=1}^{K} \nabla f_{\gamma_{t,i}}(w_t)$ is the average of the stochastic estimates of the gradient, computed by the workers, and sent to the parameter server to perform the update [17].

While the description of SSGD above uses a single data point for each gradient computation, in practice, it is common to use a mini-batch of data points, similar to Mini-Batch Gradient Descent. This is often implicitly understood, and the term 'SSGD' may be used to refer to both the single data point and mini-batch versions of the algorithm. The transition from using a single data point to a mini-batch of data points is straightforward, similar to the transition from SGD to Mini-Batch Gradient Descent (as discussed in 2.2.3).

While SSGD offers the advantage of reduced noise in the optimization process compared to ASGD, as it eliminates the issue of stale gradients, its synchronous nature can potentially slow down the training process. This is because the parameter server must wait for all workers to complete their computations before updating the model parameters. This can lead to a problem known as the *straggler effect*, where the overall computation time is dictated by the slowest worker. If one or more workers are significantly slower than the others, perhaps due to hardware issues or network latency, they can delay the entire process, leading to inefficient use of resources. Various strategies have been proposed to mitigate the straggler effect in SSGD, such as backup workers [18], gradient coding [19, 20, 21], and adaptive batch sizes [22, 23, 24]. Despite these challenges, SSGD remains a popular choice for distributed machine learning tasks due to its simplicity and robustness.

## 2.4 Federated Learning (FL)

Federated Learning (FL), as introduced in Section 1.2, is a paradigm shift in the way we approach machine learning. It addresses the need for decentralized learning in a world where data is increasingly distributed, sensitive, and often restricted in its mobility due to privacy laws and logistical constraints.

The motivation for Federated Learning arises from two main factors. First, the need to respect user privacy and data confidentiality, especially in sectors like healthcare, finance, and telecommunications. By keeping data on local devices, FL reduces the risk of data breaches and misuse. Second, the logistical constraints and legal restrictions that often prevent data from being freely moved and centralized. Federated Learning allows for the utilization of rich, diverse data sources that would otherwise be inaccessible due to these constraints.

The primary goal of FL is to train accurate and robust machine learning models while ensuring data privacy and reducing communication costs. It aims to achieve this by aggregating locally-computed updates rather than raw data, thus minimizing the amount of sensitive information transmitted. Another goal is to enable learning from non-IID (Independent and Identically Distributed) data, which is a common scenario in real-world applications where data is collected from diverse sources.

Despite its potential, FL faces several challenges. Privacy preservation, while a key advantage of Federated Learning, is also a challenge as it requires careful design to ensure that sensitive information cannot be inferred from model updates. Communication efficiency is another major challenge due to the need to transmit model updates over potentially unreliable and slow networks. Model personalization is also a concern, as the global model may not perform well for all clients due to the heterogeneity of local data. Addressing these challenges is an active area of research in Federated Learning.

In the following sections, we will delve into the technical aspects of Federated Learning, starting with the architecture, followed by a detailed exploration of the redefined objective function, and methods of Federated Learning such as Federated Stochastic Gradient Descent (FedSGD) and Federated Averaging (FedAvg).

### 2.4.1 Architecture

The architecture of a Federated Learning system is primarily composed of a central server and multiple client devices. Each client device has a local dataset that it uses to train the model, and the central server is responsible for coordinating the learning process and aggregating the model updates from the clients.

The *Central Server*, also known as *Parameter Server* or *Coordinator*, is the orchestrator of the Federated Learning process. It initiates the learning process by distributing the initial global model to the client devices. After each round of local training, the server collects the model updates from the clients, aggregates them to update the global model, and then disseminates the updated global model back to the clients for the next round of training. The server also monitors the learning process and decides when to terminate it based on the model's performance on a validation set or after a predetermined number of rounds.

The *Clients*, which can range from smartphones to IoT devices, each have a local dataset that they use to train the model. Upon receiving the global model from the server, the clients use their local data to train the model and then send the model updates back to the server. The clients do not share their raw data with the server or with each other, thus preserving data privacy. This is a key characteristic of Federated Learning, where the raw data remains on the client device and only model updates are exchanged.

The *Communication Protocol* in Federated Learning involves the exchange of model updates between the server and the clients. The server sends the global model to the clients, who then train the model on their local data and send the model updates back to the server. This process is repeated for several rounds until the learning objective is achieved. The communication protocol needs to be efficient and robust to network failures, as the clients may be connected over unreliable and slow networks.

This architecture allows Federated Learning to train machine learning models on distributed data while preserving data privacy and reducing communication costs. However, it also presents several challenges, such as ensuring privacy preservation, maintaining communication efficiency, and achieving model personalization, which are areas of ongoing research in Federated Learning.

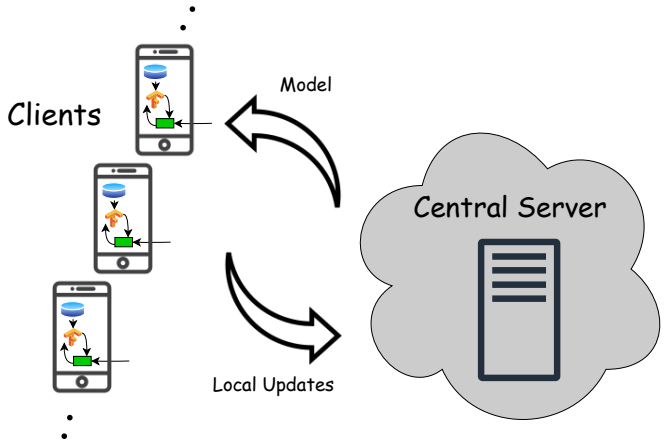



Figure 2.1: A typical architecture of a Federated Learning system. The central server coordinates the learning process, while the clients perform local training on their data. Model updates, not raw data, are exchanged between the server and the clients.

As illustrated in Figure 2.1, the architecture of a Federated Learning system is designed to facilitate decentralized learning while preserving data privacy and reducing communication costs. The central server plays a crucial role in coordinating the learning process, but the actual learning happens at the clients, each training on their own local data. This decentralization of learning is a key characteristic of Federated Learning, enabling the utilization of diverse and distributed data sources.

In this architecture, the exchange of information between the server and the clients is limited to model updates. This approach is designed to keep the raw data on the client devices, addressing the need for data privacy and logistical constraints that often prevent data from being freely moved and centralized.

The communication protocol in this architecture involves the server sending the global model to the clients, who then train the model on their local data and send the model updates back to the server. This process is repeated for several rounds until the learning objective is achieved.

### 2.4.2 Objective Function

In the context of Federated Learning (FL), the objective function takes a slightly different form compared to the traditional Stochastic Gradient Descent (SGD) setting. In SGD, as we have seen in section 2.2.2, the objective function is defined as $f(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w)$, where $n$ is the number of examples in the dataset, and $f_i(w)$ is the loss for the $i$-th example.

However, in FL, we have a network of $K$ clients, each with a unique local dataset. The data distribution across clients can be highly skewed, with some clients holding significantly more data points than others. To account for this disparity, we need to adjust our objective function.

For each client $k$, we denote $\mathcal{P}_k$ as the set of indices corresponding to its local data points, and $n_k = |\mathcal{P}_k|$ as the number of these points. We then redefine the objective function in FL as a weighted sum of the local loss functions, where the weight for each client is proportional to the number of data points it has:

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w) \quad , \text{ where } \quad F_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(w) \tag{2.2}$$

This new formulation ensures that clients with more data have a larger influence on the global model, reflecting their greater contribution to the learning process [25].

### 2.4.3 Iterative Training Process

To ensure good task performance of a final, central machine learning model, FL relies on an iterative process broken up into an atomic set of client-server interactions known as Federated Learning *rounds*. Each round consists of the following steps [4]:

1. **Client Selection:** A fraction of local clients are selected to participate in training. Selection criteria might include device availability, connection type, and idle status.
2. **Broadcast:** The selected clients download the current model weights and training program from the central server.
3. **Client Computation:** Each client computes an update to the model locally, typically using Stochastic Gradient Descent (SGD) or similar algorithms.
4. **Aggregation:** The server collects and aggregates the updates from the clients. Techniques like secure aggregation, lossy compression, and differential privacy might be applied at this stage.
5. **Model Update:** The server updates the global model based on the aggregated updates from the participating clients.

The training process in FL involves many such rounds, iteratively refining the global model. The procedure described above assumes synchronized model updates, where all selected clients complete their computations before the server aggregates the updates.

The separation of client computation, aggregation, and model update phases is not a strict requirement in FL, but it offers distinct advantages. This separation allows for a structured approach that facilitates research in areas like compression, differential privacy, and secure multi-party computation. These advances can be composed with various optimization or analytics algorithms, as long as they align with aggregation primitives. On the other hand, asynchronous approaches like ASGD (Section 2.3.2), where clients' updates are applied immediately before aggregation, are also prevalent. While these asynchronous methods may simplify system design and offer optimization benefits, they are outside the scope of this work. Instead, this section focuses on the synchronous approach, which provides a more coherent framework for integrating various research advancements.

### 2.4.4 Federated Stochastic Gradient Descent (FedSGD)

*Federated Stochastic Gradient Descent* (FedSGD) is a fundamental algorithm in the field of Federated Learning. It is a direct adaptation of the traditional Stochastic Gradient Descent (SGD) algorithm to the federated setting, where data is distributed across a large number of clients. The FedSGD algorithm is outlined below:

In the Federated Stochastic Gradient Descent algorithm (Algorithm 3) [26], each client computes the gradient over its entire dataset and sends it to the server. The server then aggregates these gradients, taking into account the size of each client's dataset, to update the global model.

**Algorithm 3** Federated Stochastic Gradient Descent (FedSGD)

**Require:** $f(w)$: Stochastic objective function with parameters $w \in \mathbb{R}^d$
**Require:** $\eta$: Step size (learning rate)
**Require:** $K$: The number of clients indexed by $k$
**Require:** $C$: Fraction of clients performing computations in each round

**Server executes:**
1: initialize $w_1$
2: **for** each round $t = 1, 2, \dots$ **do**
3: $s \leftarrow \max(C \cdot K, 1)$
4: $S_t \leftarrow$ (random set of $s$ clients)
5: **for** each client $k \in S_t$ **in parallel do**
6: $g_k \leftarrow \text{ClientUpdate}(k, w_t)$
7: $s_t \leftarrow \sum_{k \in S_t} n_k$
8: $w_{t+1} \leftarrow w_t - \eta \sum_{k \in S_k} \frac{n_k}{s_t} g_k$

**ClientUpdate($k$, $w$):** ▷ Run on client $k$
9: $g_k \leftarrow \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} \nabla f_i(w)$
10: **return** $g_k$ to server

In the context of Federated Learning, due to the distributed nature of the data and the potentially large number of clients, it's reasonable to view each client's entire dataset as a single batch. This perspective is different from the traditional setting of Stochastic Gradient Descent where a batch is a small subset of the entire dataset that resides in one place. In other words, a randomly selected client with $n_k$ training data samples in Federated Learning is analogous to a randomly selected batch in traditional machine learning. This understanding is crucial for interpreting the FedSGD algorithm.

The server does not have access to the entire dataset at once but only to the gradients computed on the individual clients' datasets which are aggregated by a weighted average. This is in accordance with the redefined objective function for FL (Equation 2.2), which is a weighted sum of the local loss functions, where the weight for each client is proportional to the number of data points it has. This ensures that each client's contribution to the global model is proportional to its data size, providing a fair and representative update, especially critical in Federated Learning where the data distribution can be highly skewed across clients.

It's worth noting that when the fraction of clients performing computations in each round, $C$, is set to 1, FedSGD becomes equivalent to full-batch SGD. In this case, all clients are selected in each round, and the server receives and aggregates gradients from all clients. This means that the server is effectively using the entire dataset (distributed across all clients) to compute the update for each round.

### 2.4.5 Federated Averaging (FedAvg)

*Federated Averaging* (FedAvg), introduced by McMahan et al. [3], is a cornerstone in the field of Federated Learning. Building upon the principles of Federated Stochastic Gradient Descent (FedSGD), FedAvg introduced a transformative concept: the application of classical Stochastic Gradient Descent, complete with epochs and mini-batches, at the client level. This seemingly simple yet profoundly impactful idea not only reduces the communication overhead but also enhances the quality of learning by allowing each client to make more effective use of its local data before sending the updated model to the server. This innovation has had a substantial impact on the field, serving as the foundation for

many subsequent advancements in Federated Learning. Despite the challenges posed by the unique characteristics of Federated Learning, such as non-IID data and unbalanced data distribution across clients, FedAvg has proven to be a robust and versatile algorithm.

**Algorithm 4** Federated Averaging (FedAvg) [3]

**Require:** $f(w)$: Stochastic objective function with parameters $w \in \mathbb{R}^d$
**Require:** $\eta$: Step size (learning rate)
**Require:** $E$: The number of local epochs
**Require:** $K$: The number of clients indexed by $k$
**Require:** $b$: The local mini-batch size
**Require:** $C$: Fraction of clients performing computations in each round

**Server executes:**
initialize $w_1$
**for** each round $t = 1, 2, \dots$ **do**
  $s \leftarrow \max(C \cdot K, 1)$
  $S_t \leftarrow$ (random set of $s$ clients)
  **for** each client $k \in S_t$ **in parallel do**
    $w_{t+1}^{(k)} \leftarrow \text{ClientUpdate}(k,\, w_t)$
  $s_t \leftarrow \sum_{k \in S_k} n_k$
  $w_{t+1} \leftarrow \sum_{k \in S_k} \frac{n_k}{s_t} w_{t+1}^{(k)}$

**ClientUpdate($k$, $w$):** ▷ Run on client $k$
Split $\mathcal{P}_k$ into batches of size $b$
**for** each local epoch $e$ from 1 to $E$ **do**
  **for** each batch $B \in \mathcal{P}_k$ **do**
    $w \leftarrow w - \eta \nabla f_B(w)$ ▷ As per Equation (2.1)
**return** $w$ to server

In the Federated Averaging algorithm (Algorithm 4), the key innovation lies in the client update step. Each client performs multiple iterations of SGD on its local data for a specified number of local epochs before sending the updated model to the server. This approach allows for more efficient use of each client's local data, enhancing the quality of learning and reducing the communication overhead between the server and the clients.

The server's role in FedAvg is to aggregate the updated models from the clients, rather than the gradients as in FedSGD. This aggregation is performed in accordance with the redefined objective function for FL (Equation 2.2), which is a weighted sum of the local loss functions, where the weight for each client is proportional to the number of data points it has. This ensures that each client's contribution to the global model is proportional to its data size, providing a fair and representative update. This is especially crucial in FL where the data distribution can be highly skewed across clients.

## 2.5 Time & Communication Costs

In the realms of Distributed Machine Learning (ML) and Federated Learning (FL), algorithmic efficiency is benchmarked against two critical measures: *CPU Time Cost*[4] and *Communication Cost*. The *CPU Time Cost* captures the parallel local computational activities during the training phase. On the other hand, the *Communication Cost* signifies

[4]The term "CPU Time Cost" encompasses aspects such as CPU processing, GPU processing, I/O operations, and memory access, with the specifics contingent upon the hardware setup and the software implementation.

data transmissions between client nodes and the central server, encompassing exchanges of model parameters (or gradients) and local state information.

Both in Distributed ML and FL, communication cost is predominantly anchored in two central operations: a reduction step and a subsequent broadcast, or vice-versa [27]. These operations have been foundational pillars in distributed systems, with extensive research efforts striving to optimize them, yielding well-defined patterns and algorithms.

### 2.5.1 All-Reduce

The *All-Reduce* pattern [27] stands as a cornerstone in traditional distributed systems, particularly within the High-Performance Computing (HPC) domain. Essential to the Message Passing Interface (MPI) libraries, this mechanism facilitates distributed units in aggregating their local values – most often gradients or model parameters in an ML context – and then disseminating the aggregated result to all participating nodes. Conceptually, All-Reduce seamlessly integrates the reduction and broadcast functions, orchestrating them in sequence, as depicted in Figure 2.2.

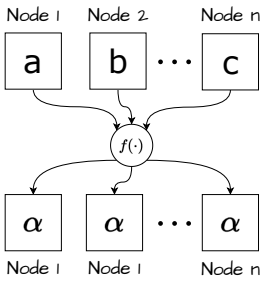



Figure 2.2: Information flow of All-Reduce operation performed on $n$ nodes, where $f(\cdot)$ is the aggregation function and $\alpha$ is the result of the reduction.

### 2.5.2 Unification of costs

In the quest to provide a holistic analysis of algorithmic efficiency in Distributed ML and FL, it's paramount to have a singular metric that encapsulates both *CPU Time Cost* and *Communication Cost*. Thus, a unified metric, termed as the *Time Cost*, emerges:

$$(\text{Time Cost}) = (\text{CPU Time Cost}) + (\text{Communication Time Cost}) \tag{2.3}$$

The *Communication Time Cost* is derived through abstract approaches, reflecting the underlying communication channels, topology, and other factors. These approaches are not drawn from specific literature but are based on general principles of distributed systems. The primary objective is to construct a coherent framework for contrasting different methodologies rather than delineating precise models. By doing so, this framework empowers us to evaluate various strategies on a singular, cohesive time scale.

For both communication models described below, let $C$ symbolize the total communication cost in Gbit, $B$ be the channel bandwidth in Gbps, and $K$ be the number of clients (or nodes).

**Common Channel Communication Model:** This model aims to capture the dynamics in a distributed system where all clients share a single communication channel to access the central server. The inherent nature of this shared resource can induce delays, exacerbated as the client count increases.

$$(\text{Communication Time Cost}) = (K-1) \cdot \frac{C}{K} \cdot \frac{1}{B} \tag{2.4}$$

**Hypercube Communication Model:** In a hypercube topology, the communication time cost decreases logarithmically as the number of clients increases.

$$(\text{Communication Time Cost}) = \lceil \log K \rceil \cdot \frac{C}{K} \cdot \frac{1}{B} \tag{2.5}$$

In High-Performance Computing (HPC) environments, hypercube-based designs can be advantageous, especially when reducing latency is pivotal, like in All-Reduce operations.

## 2.6 Artificial Neural Networks (ANNs)

In this section, we delve into the fascinating world of Artificial Neural Networks (ANNs), the driving force behind many modern machine learning systems. ANNs have revolutionized fields ranging from image recognition to natural language processing, and understanding their inner workings is essential for any researcher in the field of machine learning.

In the forthcoming subsections, we embark on an exploration of the fundamental building blocks of Neural Networks, specifically focusing on the structure and operation of Multi-Layer Perceptrons (MLPs), which are composed of multiple fully-connected layers of neurons. This exploration is guided by the approach adopted by Professor Karystinos in his lectures on Neural Networks [28]. His intuitive approach allows us to convey the essential concepts in a clear and accessible manner, circumventing the often confusing notation prevalent in the general literature on the subject.

We then extend our discussion to Convolutional Neural Networks (CNNs), a specialized kind of neural network that has proven exceptionally effective in tasks related to image processing. We also delve into the learning process in these networks, focusing on the backpropagation algorithm and optimization methods. Along the way, we highlight some of the challenges encountered when training deep networks and present strategies for mitigating these issues.

### 2.6.1 Perceptron & Neuron

To understand the concept of a Perceptron and a Neuron, let's start with a simple dataset $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where each $x_i$ is a $d$-dimensional vector and each $y_i$ is a real number.

A neuron is a computational unit that takes as input a vector $x_i$ and produces an output (or prediction) $\hat{y}$, based on a set of parameters $(w, b)$ and a nonlinear function $\sigma(\cdot)$, known as the *activation function*. The parameters $w$ and $b$ are a weight vector and a bias term, respectively. The weight vector is the same dimension as the input vector, and the bias is a single real number.

The output of the neuron is computed as follows:

$$\hat{y} = \sigma\left(b + \sum_{i=1}^{d} w_i x_i\right) = \sigma\left(w^\top x + b\right)$$

This equation represents the neuron taking a weighted sum of its inputs (plus the bias term), and then applying the activation function to this sum. The activation function introduces non-linearity into the model, allowing it to learn more complex patterns. The structure of a single neuron is depicted in Figure 2.3.

A *Perceptron* is a special case of a neuron, where the activation function is a threshold function. This means that the Perceptron outputs a binary value, depending on whether the weighted sum of its inputs exceeds a certain threshold [29, 30].

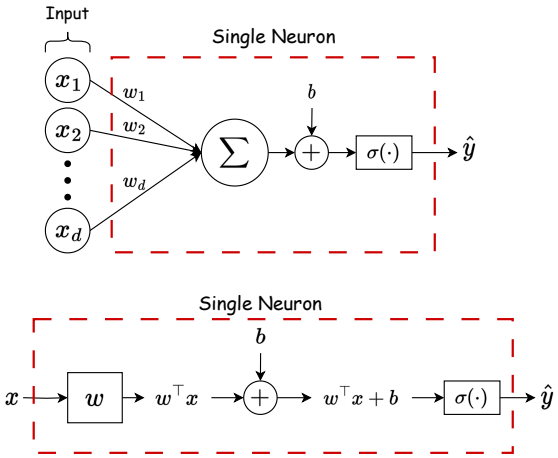



Figure 2.3: Two equivalent representations of a single neuron

In the next section, we will extend this concept to a network of neurons, forming a Multi-Layer Perceptron (MLP).

### 2.6.2 Multi-Layer Perceptron (MLP)

Building upon the concept of a single neuron, we can create a more complex structure by stacking multiple neurons together. This leads us to the concept of a fully-connected layer.

In a fully-connected layer, we have a set of neurons, each of which is connected to all inputs. Let's consider our dataset $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where each $x_i$ is a $d$-dimensional vector and each $y_i$ is a $p$-dimensional vector. This is a slight departure from our previous discussion where $y_i$ was a scalar. Now, we are considering the case where the output could be a vector, allowing us to handle multi-output problems.

In this layer, each neuron has its own set of parameters $(w_i, b_i)$, where $w_i \in \mathbb{R}^d$ and $b_i \in \mathbb{R}$, and uses the same activation function $\sigma(\cdot)$. The output of this layer is a vector $\hat{y} \in \mathbb{R}^p$, where each element is the output of a neuron:

$$\hat{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix} = \begin{bmatrix} \sigma\left(w_1^\top x + b_1\right) \\ \sigma\left(w_2^\top x + b_2\right) \\ \vdots \\ \sigma\left(w_p^\top x + b_p\right) \end{bmatrix} = \sigma\left( \begin{bmatrix} w_1 & w_2 & \ldots & w_p \end{bmatrix}^\top x + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{bmatrix} \right) = \sigma\left(W^\top x + b\right)$$

Here, $W \in \mathbb{R}^{p \times d}$ is a matrix where each column is a weight vector of a neuron, and $b \in \mathbb{R}^p$ is a vector where each element is the bias of a neuron. Furthermore, the activation function $\sigma(\cdot)$ is applied element-wise. The structure of a fully-connected layer is depicted in Figure 2.4.

A remarkable property of a fully-connected layer, and indeed of any layer in a neural network, is that its entire functionality can be encapsulated by the weight matrix $W$ and the bias vector $b$. This succinct representation not only provides a clear and concise description of the layer's operation but also facilitates efficient computation.

Now, imagine stacking multiple fully-connected layers one after the other. The output of one layer becomes the input to the next. This structure is known as a *Multi-Layer Perceptron* (MLP). Despite the term "Perceptron" suggesting a network composed solely of Perceptrons, the term 'MLP' is used more broadly to denote any fully-connected feed-forward network, regardless of the specific activation functions used. This is more of a

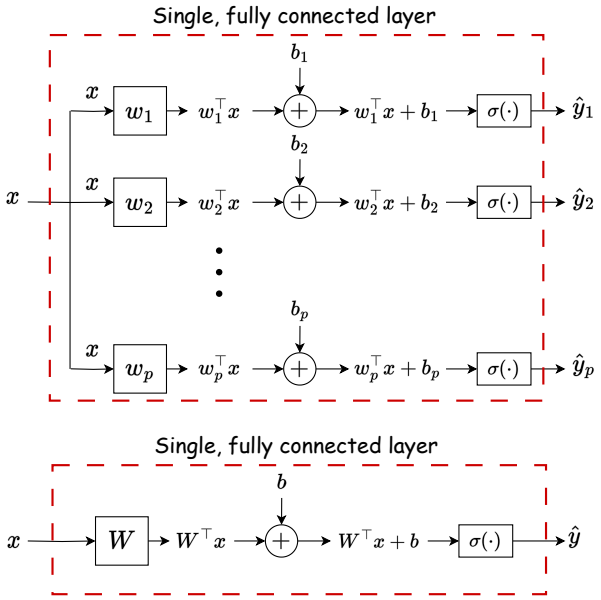



Figure 2.4: Two equivalent representations of a single, fully-connected layer

historical artifact. The layers in between the input and output layers are often referred to as *hidden layers*, as they are not directly exposed to the input or output of the network. This architecture allows the network to learn more complex representations of the input data, as each layer can potentially capture different features or patterns.

### 2.6.3 Convolutional Neural Networks (CNNs)

*Convolutional Neural Networks (CNNs)* are a specialized kind of neural network that are exceptionally effective at tasks related to image processing, such as image classification, object detection, and semantic segmentation. The defining characteristic of CNNs is the use of convolutional layers, which are designed to automatically and adaptively learn spatial hierarchies of features from the input data.

CNNs can be viewed as a regularized version of Multi-layer Perceptrons (MLPs). Recall that in an MLP (Section 2.6.2), each neuron in one layer is connected to all neurons in the next layer. In contrast, CNNs connect each neuron only to a local region of the input, referred to as the neuron's *receptive field*. This difference in architecture reduces the number of parameters in the model, as each filter is typically much smaller than the input data, and provides a degree of translation invariance [5], as the same filter is applied across the entire input. By limiting the complexity of the model in this way, CNNs help prevent overfitting, a common problem in fully connected networks.

In a convolutional layer, the weights of the neuron, also known as a *filter* or *kernel*, are convolved with the input data, producing a *feature map*. This process is repeated with multiple filters, each producing a separate feature map, and the results are stacked together to form the output of the layer. This operation allows the model to learn local features, which are often more meaningful in image data.

Furthermore, CNNs exploit the hierarchical structure in data, building on simpler features to form more intricate representations. Instead of processing the entire image or input at once, CNNs dissect it into smaller, simpler features. As the network progresses through the layers, these features are combined and assembled into more sophisticated

[5] This refers to the property that the network's detection of a particular feature is not dependent on where in the image that feature is located. In other words, the network can recognize the same feature regardless of its position in the input space.

structures, allowing the network to learn increasingly abstract representations of the input. This hierarchical approach allows CNNs to efficiently learn complex patterns in data.

In addition to convolutional layers, CNNs often include other types of layers, such as *pooling layers*, which reduce the spatial dimensions of the data, and fully-connected layers, which are typically used at the end of the network for classification or regression tasks.

CNNs have been the driving force behind many of the recent advances in image processing and computer vision, and they continue to be a topic of active research. For more detailed information on CNNs, we refer the reader to the seminal works by LeCun et al. [31] and Krizhevsky et al. [32].

### 2.6.4 Learning: Backpropagation & Optimization

The goal of learning in neural networks is to find the set of parameters (weights and biases) that minimize a *loss function* (or *objective function* as we saw in Sections 2.2, 2.3, 2.4). The loss function quantifies the discrepancy between the network's predictions and the actual target values. If we had the gradient of the loss function with respect to the parameters, we could easily update the parameters using one of the optimization algorithms we've previously covered, such as Stochastic Gradient Descent (Section 2.2.2), Mini-batch Gradient Descent (Section 2.2.3) etc.

However, the challenge lies in efficiently computing these gradients, especially for deep networks with many layers and a large number of parameters. This is where the *backpropagation* algorithm comes into play. It was first introduced in 1960s [33] and almost 30 years later (1989) popularized by Rumelhart, Hinton and Williams in [34]. Backpropagation is a method for efficiently computing the gradient of the loss function with respect to the parameters of the network. The algorithm is based on the chain rule from calculus and operates in two passes through the network: a forward pass and a backward pass.

In the forward pass, the input is propagated through the network to compute the output and the loss. In the backward pass, the algorithm works backward from the output layer to the input layer, propagating the error information back into the network. For each layer, the algorithm computes the gradient of the loss with respect to the parameters of that layer.

This process of forward propagation, backpropagation, and parameter update is repeated iteratively over multiple epochs until the network's performance on the data reaches a satisfactory level or until some stopping criterion is met.

Despite the power of backpropagation and gradient-based optimization, these methods are not without their challenges. Two common issues in training deep neural networks are the *vanishing gradients* and *exploding gradients* problems. The vanishing gradients problem occurs when the gradients of the loss function become very small as they are propagated back through the network. This can slow down learning or cause it to stop altogether, as the updates to the weights become negligibly small. The exploding gradients problem, on the other hand, occurs when the gradients become very large, leading to large updates to the weights and potentially causing the model to diverge. These problems are particularly pronounced in deep networks and recurrent neural networks, and they are influenced by factors such as the choice of activation function, the initialization of the weights, and the architecture of the network. Various techniques have been proposed to mitigate these problems. For instance, the use of activation functions like ReLU that are less prone to vanishing gradients was proposed by Nair and Hinton [35], careful initialization strategies were suggested by Glorot and Bengio [36], and normalization techniques like batch normalization were introduced by Ioffe and Szegedy [37].

In summary, backpropagation is a fundamental building block in a neural network, enabling the efficient computation of gradients. Coupled with an optimization algorithm, it forms the backbone of learning in neural networks, allowing the network to iteratively

adjust its parameters to minimize the loss and make accurate predictions on the data. However, the training process is not always straightforward and can present challenges that require specific solutions. For a comprehensive understanding of these topics, including an in-depth discussion of the challenges encountered in training deep neural networks and the various strategies proposed to mitigate them, we refer the reader to the work by Goodfellow et al. [38].

## 2.7 Frameworks for Deep Learning

Deep learning, has been pivotal in driving advancements across a myriad of domains, from computer vision and natural language processing to medical diagnostics and autonomous systems. At the heart of these advancements lie powerful computational frameworks that facilitate model design, training, and deployment. Among these, TensorFlow, complemented by its high-level API, Keras, has gained widespread adoption in both academia and industry. While several formidable frameworks mark the landscape of deep learning, this section focuses on TensorFlow and Keras, which were chosen for their alignment with our requirements in the creation of the Simulator later discussed in Section 4.1.

### 2.7.1 TensorFlow

TensorFlow is an open-source machine learning platform designed for efficiency, scalability, and flexibility [39, 40]. Its foundational capabilities include:

- **Versatile Execution**: TensorFlow can efficiently manage tensor operations across a variety of hardware, from CPUs and GPUs to TPUs. Its design is rooted in the computation graph paradigm, where mathematical operations are nodes and multi-dimensional arrays (tensors) are the edges, ensuring efficient computation and gradient calculations.
- **Deep Learning with Keras**: Within the TensorFlow framework, Keras acts as the high-level API, making deep learning accessible and intuitive. This combination allows users to design sophisticated models using Keras, all while harnessing the computational prowess of TensorFlow—be it on TPU clusters, GPU arrays, or other configurations.
- **Flexibility in Model Building**: Whether a beginner or an expert, TensorFlow caters to all with its layered abstraction. While the high-level Keras API offers simplicity, eager execution provides a more hands-on approach for immediate model iteration and debugging.
- **Deployment Everywhere**: TensorFlow streamlines the path from development to production, allowing models to be trained and deployed across servers, edge devices, or web platforms. Its versatility ensures that regardless of the target platform, deploying a TensorFlow model remains consistent and straightforward.
- **Research and Experimentation**: TensorFlow isn't just a tool—it's a playground for research. Advanced features like the Keras Functional API and Model Subclassing API let users create complex model architectures. Coupled with the vibrant ecosystem of libraries such as TensorFlow Probability and BERT [41], it's primed for state-of-the-art research.

In essence, TensorFlow is a robust platform for everything from simple model development to groundbreaking machine learning research, offering tools that cater to a wide spectrum of needs and complexities.

### 2.7.2 Keras

Keras, developed in Python, is a deep learning API that stands at the forefront of the TensorFlow machine learning platform [42]. Designed with a clear focus on modern deep learning, Keras provides an approachable and highly-productive interface for tackling machine learning (ML) challenges. It addresses every phase of the machine learning workflow, spanning data processing, hyperparameter tuning, and deployment. Keras epitomizes the principle of swift experimentation—turning ideas into results efficiently, a fundamental aspect of state-of-the-art research. Key attributes of Keras include:

- **Simplicity**: By abstracting underlying complexities, Keras minimizes developers' cognitive load, allowing them to zero in on central aspects of the problem.
- **Power and Scalability**: Despite its streamlined interface, Keras doesn't compromise on performance. This balance between ease-of-use and power has led to its adoption by renowned organizations, including Netflix, YouTube, and Uber.
- **Research Excellence**: Keras's flexibility caters to both fundamental tasks and pioneering research ideas. Its adaptability is reflected in its utilization by prestigious scientific institutions worldwide, such as CERN, NASA, and NIH, playing a role even in groundbreaking experiments at the LHC.

While TensorFlow provides the foundational infrastructure, Keras stands as its high-level API, furnishing an intuitive yet productive interface tailored for machine learning challenges, especially in the domain of deep learning.

# Chapter 3

# Functional Dynamic Averaging

In this chapter, we introduce the Functional Dynamic Averaging (FDA) synchronization strategy, which is specifically tailored for Server-based architectures. Originally conceived for application in the Distributed Online ML setting [43], the FDA strategy addresses scenarios where training is protracted, and the influx of training data varies over time. Particularly in environments where computational resources, such as network bandwidth, are shared with other tasks, the FDA strategy aims to minimize the communication overhead during training.

The adaptability of the FDA strategy allows for its seamless integration into the Federated Learning (FL) setting with only minor modifications. In this chapter, we will elucidate its application in the FL context, including the incorporation of additional server functionality into the FL round, as detailed in Section 2.4.3. Readers can infer that all concepts and methodologies discussed herein are derived from the work in [43], thus eliminating the need for repetitive referencing.

In Chapter 2, we have laid the groundwork for the discussions that follow. This groundwork ensures that any notation, concept, or definition not introduced in later discussions has been clearly defined beforehand. As we progress, we will build upon these foundational elements and introduce new concepts as needed, referencing previous material to foster a coherent exploration of the subject matter.

## 3.1 Setting & Notation

In this subsection, we briefly restate the notation and terminology relevant to the standard Federated Learning (FL) setting, and introduce additional concepts that will be utilized in the discussions that follow. The goal is not to delve into detailed explanations, as most of these have been covered in Section 2.4, but to offer readers a convenient reference, consolidating all pertinent information in one place for ease of access.

Consider a FL system with $K$ clients. For each client $k$, we define $\mathcal{P}_k$ as the set of indices corresponding to its local data points, and $n_k = |\mathcal{P}_k|$ as the count of these points. The total number of data points across all clients is denoted by $n$. At any given time $t$, each client $k$ maintains a model with $d$-dimensional parameters, represented by $w_t^{(k)}$.

The system's objective is to minimize a loss function $f(w)$, defined as:

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w) \quad \text{, where} \quad F_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(w)$$

At time $t$, let $w_{t_0}$ be the model [1] last sent to the clients by the parameter server, and

[1] Throughout this work, the terms "model" and "model parameters" may be used interchangeably. This shorthand is common in the literature and is adopted here for brevity.

$\overline{w}_t$ be the weighted average model of the clients. We introduce $\Delta_t^{(k)}$ as the change to the local model at time $t$ since the beginning of the current round at time $t_0$, and $\overline{\Delta}_t$ as the weighted average change. These are defined as:

$$\overline{w}_t = \sum_{k=1}^{K} \frac{n_k}{n} w_t^{(k)} \tag{3.1}$$

$$\Delta_t^{(k)} = w_t^{(k)} - w_{t_0} \tag{3.2}$$

$$\overline{\Delta}_t = \sum_{k=1}^{K} \frac{n_k}{n} \Delta_t^{(k)} \tag{3.3}$$

## 3.2 Intuition

In this section, we step away from the formalities and delve into the intuitive understanding of the Federated Learning training process. Through an engaging analogy and a closer look at the underlying challenges, we aim to provide a clear and relatable insight into the complex dynamics at play. This exploration serves as a foundation, shedding light on the reasoning behind the approach we take in this chapter, and setting the stage for the technical details that follow.

The Federated Learning training process embarks on a complex quest to discover a local minimum that serves the global model effectively, recognizing that finding a good enough local minimum is often the best one can aspire to. This task is complex, given that models often reside in extremely high-dimensional spaces filled with numerous local minima, many of which could be suitable solutions, but are often vastly separated from one another. Even if individual clients were to possess satisfactory model parameters, aggregating them can lead to a meaningless model—a random point in the space with poor accuracy. The challenge, then, is to coordinate the clients so they can cooperatively navigate this high-dimensional space along a roughly similar path. This coordination is achieved through an iterative, round-based approach (Section 2.4.3). In each round, the server broadcasts the global model, clients train locally, and the server then aggregates the models. The process is akin to a shared journey, where all participants start from the same point, follow the gradients to explore and progress through the landscape, and then combine their discoveries to set a new common starting point for the next iteration. To better illustrate this complex process, let's consider an analogy that captures the essence of the FL iterative training process:

Consider a classroom filled with children and a teacher who has tasked them with collaboratively painting a house on a canvas. Each child has a specific section of the canvas to work on. They begin with a shared goal, but as they paint, each part appears unconnected and almost random, regardless of the beauty of each piece. The teacher soon observes this and halts their progress, pointing out that they've strayed from the shared objective: creating a beautiful depiction of a house. After assessing the current state of the artwork, the teacher outlines a strategy to unify the disparate elements, and the children resume painting. Inevitably, they once again become absorbed in their individual tasks, losing sight of the collective goal. The teacher, recognizing the recurring issue, redirects them towards the common vision and offers guidance on weaving their separate contributions into a cohesive whole. This cycle repeats, but as the painting evolves and the image of the house takes shape, the children's focus on the shared goal sharpens. With the beautiful house now clearly emerging before them, and no alternative paths to distract them, they find themselves needing less and less guidance to bring the artwork to completion.

The classroom painting scenario offers a striking analogy to the Federated Learning

(FL) training process. In FL, the server acts as the guiding teacher, coordinating the efforts of clients (the children) as they navigate the high-dimensional space of model parameters. The clients in FL, like the children, begin with a common model, but as they train or paint individually, especially at the beginning, their efforts diverge, resulting in disparate and unconnected outcomes. The iterative, round-based approach refines and aligns these models, akin to the teacher's guidance in aligning the children's work towards the shared goal of a beautiful house. The process continues, with the clients converging towards a common path in the high-dimensional space, much like the children seeing the painting take shape and recognizing the one clear direction to complete it.

There exists a key distinction between the classroom painting scenario and the Federated Learning (FL) training process, a distinction that will be the focal point in the sections that follow. In the classroom, the teacher monitors the progress of the canvas, intervening to halt the children's painting and redirect them toward the common goal only when their individual parts have drifted apart. If their work aligns, the teacher allows them to continue without interruption. In contrast, in FL, as exemplified by methods like FedSGD (Section 2.4.4) and FedAvg (Section 2.4.5), the rounds are terminated at predetermined intervals, such as after a single large local step in FedSGD or some local epochs in FedAvg. This round termination occurs on a fixed schedule, regardless of whether the clients' models are harmonious and on the same path. Moreover, especially at the start, as clients navigate the high-dimensional space with no clear common goal, each following a path to a different local minimum, rounds must terminate more frequently to set the clients on a common path. This is akin to the early stages of the children's painting, where the image of the house is not yet apparent, and they may quickly lose focus or stray in different directions. The teacher must intervene more often to guide them back to a shared vision. In FL, it's this rigid, often unnecessary round termination, coupled with the occasional need for more frequent terminations to realign divergent paths, that we aim to address and alleviate. We recognize these as potential bottlenecks in terms of communication and time cost in the FL training process.

However, there is another nuanced distinction between the two scenarios that sets them apart. In the classroom, the teacher has a collective view of the progress at all times. A simple glance at the canvas reveals whether the children's individual efforts have drifted from the common goal. In the FL training process, the server lacks this collective view. To understand if the clients' models have drifted apart and if the round should be terminated, the server would have to centralize all the client models for aggregation. This centralization is exactly what we aim to circumvent. Recognizing this challenge, in the next section, we will introduce a measure known as the weighted-variance. This measure serves as a gauge, indicating to the server if the clients' models have begun to drift apart. In the following sections, we will explore how the server can monitor this measure, allowing it to take a "sneak peek" at the state of the training with minimal cost, and providing insight into whether the clients are working collectively towards a shared goal. By doing so, the server can make informed decisions about round termination without centralizing and aggregating the models, reducing unnecessary communication and keeping the training process aligned with the principles of Federated Learning.

## 3.3 Model Variance

In the context of FDA in the Federated Learning setting, understanding the variance among the models maintained by different clients is crucial. The concept of *weighted model variance* quantifies the dispersion or spread of the client models around the weighted average model. At time $t$, it is defined as:

$$\sum_{k=1}^{K} \frac{n_k}{n} \left\| w_t^{(k)} - \overline{w}_t \right\|_2^2$$

This measure provides insight into how closely aligned the clients' models are at any given time. A high variance indicates that the models are widely spread out, possibly leading to a lack of cohesion in the aggregated model. Conversely, a low variance suggests that the clients' models are closely aligned, working collectively towards the shared objective. The weighted model variance plays a critical role in our approach, as it helps the server to gauge the state of the training process and make informed decisions about round termination and other aspects of the training process.

In the FL setting, the data distribution across clients is often highly skewed. As a result, the models of clients with more data must be given more consideration than those from clients with less data. This is why taking the weighted average is standard practice, and the variance is no exception to this rule. The aggregated global model must align more closely with the models from clients with more data and less so with models from clients with less data. This approach ensures that the aggregated model reflects a meaningful point in the parameter space, taking into account the underlying data distribution.

**Theorem 1** *The weighted model variance can be written as*

$$\sum_{k=1}^{K} \frac{n_k}{n} \left\| w_t^{(k)} - \overline{w}_t \right\|_2^2 = \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} \right\|_2^2 \right) - \left\| \overline{\Delta}_t \right\|_2^2$$

**Proof:**

$$\begin{aligned}
\sum_{k=1}^{K} \frac{n_k}{n} \left\| w_t^{(k)} - \overline{w}_t \right\|_2^2 &= \sum_{k=1}^{K} \frac{n_k}{n} \left\| w_t^{(k)} - w_{t_0} + w_{t_0} - \overline{w}_t \right\|_2^2 \overset{(3.3)}{=} \overset{(3.2)}{=} \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} - \sum_{k=1}^{K} \frac{n_k}{n} \Delta_t^{(k)} \right\|_2^2 = \\
&\overset{(3.3)}{=} \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} - \overline{\Delta}_t \right\|_2^2 = \sum_{k=1}^{K} \frac{n_k}{n} \left( \left\| \Delta_t^{(k)} \right\|_2^2 - 2 \left\langle \Delta_t^{(k)} , \overline{\Delta}_t \right\rangle + \left\| \overline{\Delta}_t \right\|_2^2 \right) = \\
&= \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} \right\|_2^2 \right) - 2 \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\langle \Delta_t^{(k)} , \overline{\Delta}_t \right\rangle \right) + \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \overline{\Delta}_t \right\|_2^2 \right) = \\
&= \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} \right\|_2^2 \right) - 2 \left\langle \sum_{k=1}^{K} \frac{n_k}{n} \Delta_t^{(k)} , \overline{\Delta}_t \right\rangle + \left\| \overline{\Delta}_t \right\|_2^2 = \\
&\overset{(3.3)}{=} \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} \right\|_2^2 \right) - 2 \left\langle \overline{\Delta}_t , \overline{\Delta}_t \right\rangle + \left\| \overline{\Delta}_t \right\|_2^2 = \\
&= \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} \right\|_2^2 \right) - 2 \left\| \overline{\Delta}_t \right\|_2^2 + \left\| \overline{\Delta}_t \right\|_2^2 = \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} \right\|_2^2 \right) - \left\| \overline{\Delta}_t \right\|_2^2
\end{aligned}$$

## 3.4 Round Terminating Condition (RTC)

Building on the concept of weighted model variance, we introduce a specific condition to determine when to terminate a round in the Federated Learning process. This condition, referred to as the *Round Terminating Condition*, plays a vital role in enhancing the efficiency of the training process.

We define $\Theta$, a hyper-parameter of the FDA, at the beginning of each round; it may change at each round. When the model variance exceeds this value, the FL round must

terminate. As long as it is below this threshold, local training on the clients continues. The Round Terminating Condition (RTC) is expressed as:

$$\sum_{k=1}^{K} \frac{n_k}{n} \left\| w_t^{(k)} - \overline{w}_t \right\|_2^2 \leq \Theta \quad \underset{\text{Theorem 1}}{\Leftrightarrow} \quad \left( \sum_{k=1}^{K} \frac{n_k}{n} \left\| \Delta_t^{(k)} \right\|_2^2 \right) - \left\| \overline{\Delta}_t \right\|_2^2 \leq \Theta \tag{3.4}$$

## 3.5 Monitoring the Round Terminating Condition (RTC)

At this juncture, our focus shifts to devising a method for the server to monitor the RTC, as defined in Equation (3.4). The challenge lies in adapting techniques from the well-established field of distributed stream monitoring to the specific requirements of the Federated Learning (FL) setting. To align with the general literature, we will restate the problem of monitoring the RTC using the standard distributed stream monitoring formulation. This alignment allows us to leverage optimal methods such as Functional Geometric Monitoring [7].

We begin by defining the *local state* for each client $k$. The local state, denoted by $S_k(t)$, encapsulates information from client $k$ for the monitoring task at time $t$:

$$S_k(t) \in \mathbb{R}^p, \quad k = 1, \ldots, K$$

The local state is updated arbitrarily, reflecting the client's local data and computations.

Next, we introduce the *global state*, $S(t)$, which represents the collective state of the Federated Learning system at time $t$. It is defined as the weighted average of the local states:

$$S(t) = \sum_{k=1}^{K} \frac{n_k}{n} S_k(t) \tag{3.5}$$

The monitoring task aims to observe a threshold condition on the global vector, expressed as:

$$F(S(t)) \leq \Theta \tag{3.6}$$

where $F : \mathbb{R}^p \to \mathbb{R}$ is a non-linear function.

While the above definitions may appear abstract, they will be concretized in subsequent sections. We begin with the following theorem:

**Theorem 2** *Define* $S_k(t) = \left[ \left\| \Delta_t^{(k)} \right\|_2^2 \quad \Delta_t^{(k)} \right]^\top \in \mathbb{R}^{d+1}$ *and* $F(v, x) = v - \|x\|_2^2$. *The condition* $F(S(t)) \leq \Theta$ *is equivalent to the RTC.*

**Proof:** Follows directly from Equation (3.4).

Unfortunately, the Functional Geometric Monitoring (FGM) method faces a challenge in monitoring the RTC (as outlined in Theorem 2) with low communication cost. This difficulty arises from the high dimensionality of $S_k$, where $d + 1$ could be in the order of millions to billions. Such high dimensions are particularly common in widely used models like deep neural networks, making the application of FGM in this context a complex task. To mitigate this communication burden, we must apply dimensionality reduction to the local model changes $\Delta^{(k)}$ and identify an appropriate function to monitor based on the chosen reduction.

However, this reduction in information within the local state vectors causes monitoring the RTC to become approximate. A tradeoff emerges between the performance of

the monitoring task and the tightness of the approximation. In the following sections, we will introduce three specific reduction methods for monitoring the RTC with minimal communication. These methods, referred to as the "naive," "linear," and "sketch" approximations, provide different ways to approximate the local state vectors.

We now turn our attention to devising the general FDA algorithm for FL iterative training with monitoring the RTC approximately. The details of the algorithm, which integrates the concepts discussed earlier, are presented below:

**Algorithm 5** FDA: FL training with Approximate RTC Monitoring [43]

**Require:** $f(w)$: Stochastic objective function with parameters $w \in \mathbb{R}^d$
**Require:** $K$: The number of clients indexed by $k$
**Require:** $\Theta$: Monitoring threshold
**Require:** $S_k(t)$: The local state for client $k$ where $S_k(t) \in \mathbb{R}^p$ and $p \ll d$
**Require:** $F(x)$: A function $F : \mathbb{R}^p \to \mathbb{R}$ such that $F(S(t)) \leq \Theta$ implies the RTC
**Require:** $\eta$: Step size (learning rate)
**Require:** $s$: The number of local steps
**Require:** $b$: The local mini-batch size

**Server executes:**
1: initialize $w_1$
2: **for** each round $t = 1, 2, \ldots$ **do**
3: Broadcast $w_t$ to all clients
4: **repeat**
5: **for** each client $k = 1, \ldots, K$ **in parallel do**
6: $S_k(t) \leftarrow \text{ClientTrain}(k)$
7: **until** $F(S(t)) > \Theta$ ▷ As per Equation (3.5)
8: **for** each client $k = 1, \ldots, K$ **in parallel do**
9: $w_{t+1}^{(k)} \leftarrow$ (download the model of client $k$)
10: $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^{(k)}$

**ClientTrain($k$):** ▷ Run on client $k$
11: $\mathcal{B} \leftarrow$ (Choose $s$ batches of size $b$ from $\mathcal{P}_k$)
12: **for** each batch $B \in \mathcal{B}$ **do**
13: $w \leftarrow w - \eta \nabla f_B(w)$ ▷ As per Equation (2.1)
14: **return** $S_k(t)$ to server

To clarify the context of Algorithm 5, the expression "$F(S(t)) \leq \Theta$ implies the RTC" indicates that if the condition $F(S(t)) \leq \Theta$ is met, then the RTC condition of Equation (3.4) is also satisfied. For the sake of simplicity, we assume that $\Theta$ is fixed throughout all rounds. While this assumption may not reflect all practical scenarios, it serves our analytical purpose in this context. The algorithm primarily employs Stochastic Gradient Descent (SGD) for optimization, but it can be easily adapted to accommodate other optimization methods. In the algorithm, we posit that all clients participate in every round. This departure from the classic FL round definition, as presented in Section 2.4.3, serves a specific analytical purpose. Ensuring uniform participation simplifies the discourse and allows us to focus on the core mechanics of monitoring the Round Terminating Condition (RTC). The approach facilitates exploration of specific aspects such as communication efficiency and dimensionality reduction, without the complexity of client selection. Most importantly, it serves as a basis for our methodological exploration, and the insights gained can be extended to more generalized client selection strategies. It's a deliberate modelling choice that, within the scope of this work, does not compromise the validity or applicability

of our findings.

## 3.6 Naive Approximation

In the naive approach, we eliminate the $\Delta^{(k)}$ vector from the local state, i.e., reduce its dimension to zero [43].

**Theorem 3 (Naive FDA)** *Define* $S_k(t) = \left\|\Delta_t^{(k)}\right\|_2^2 \in \mathbb{R}$. *Also, define* $F(v) = v$. *Then, the condition* $F(S(t)) \leq \Theta$ *implies the RTC.*

**Proof:**

$$F(S(t)) \overset{(3.5)}{=} F\left(\sum_{k=1}^{K} \frac{n_k}{n} S_k(t)\right) = \sum_{k=1}^{K} \frac{n_k}{n} \left\|\Delta_t^{(k)}\right\|_2^2 \geq \left(\sum_{k=1}^{K} \frac{n_k}{n} \left\|\Delta_t^{(k)}\right\|_2^2\right) - \left\|\overline{\Delta}_t\right\|_2^2 \quad (3.7)$$

We proved in (3.7) that $F(S(t))$ always overestimates the actual variance (Theorem 1). Hence, the condition $F(S(t)) \leq \Theta$ implies the RTC, completing the proof [43].

## 3.7 Linear Approximation

In the linear approach, we reduce the $\Delta^{(k)}$ vector to a scalar, $\left\langle \xi\,,\, \Delta_t^{(k)} \right\rangle \in \mathbb{R}$ where $\xi$ is any unit vector [43].

**Theorem 4 (Linear FDA)** *Define* $S_k(t) = \left[\left\|\Delta_t^{(k)}\right\|_2^2 \quad \left\langle \xi\,,\, \Delta_t^{(k)} \right\rangle\right]^\top \in \mathbb{R}^2$ *where* $\|\xi\|_2 = 1$. *Also, define* $F(v, x) = v - x^2$. *Then, the condition* $F(S(t)) \leq \Theta$ *implies the RTC.*

**Proof:**

$$F(S(t)) \overset{(3.5)}{=} F\left(\sum_{k=1}^{K} \frac{n_k}{n} S_k(t)\right) = \left(\sum_{k=1}^{K} \frac{n_k}{n} \left\|\Delta_t^{(k)}\right\|_2^2\right) - \left(\sum_{k=1}^{K} \frac{n_k}{n} \left\langle \xi\,,\, \Delta_t^{(k)} \right\rangle\right)^2 =$$

$$= \left(\sum_{k=1}^{K} \frac{n_k}{n} \left\|\Delta_t^{(k)}\right\|_2^2\right) - \left|\left\langle \xi\,,\, \sum_{k=1}^{K} \frac{n_k}{n} \Delta_t^{(k)} \right\rangle\right|^2 = \left(\sum_{k=1}^{K} \frac{n_k}{n} \left\|\Delta_t^{(k)}\right\|_2^2\right) - \left|\left\langle \xi\,,\, \overline{\Delta}_t \right\rangle\right|^2 \geq$$

$$\overset{(3.9)}{\geq} \left(\sum_{k=1}^{K} \frac{n_k}{n} \left\|\Delta_t^{(k)}\right\|_2^2\right) - \left\|\overline{\Delta}_t\right\|_2^2 \quad (3.8)$$

$$\left|\left\langle \xi\,,\, \Delta_t^{(k)} \right\rangle\right|^2 \leq \|\xi\|_2^2 \left\|\overline{\Delta}_t\right\|_2^2 = 1 \left\|\overline{\Delta}_t\right\|_2^2 = \left\|\overline{\Delta}_t\right\|_2^2 \quad \text{(Cauchy–Schwarz)} \quad (3.9)$$

We proved in (3.8) that $F(S(t))$ always overestimates the actual variance (Theorem 1). Hence, the condition $F(S(t)) \leq \Theta$ implies the RTC, completing the proof [43].

A random choice of $\xi$ is likely to perform poorly (terminate a round prematurely), as it will likely be close to orthogonal to $\overline{\Delta}_t$. A good choice would be a vector correlated to $\overline{\Delta}_t$. A heuristic choice is to take $\overline{\Delta}_{t_0}$, i.e., the change vector right before the current round started. All nodes can estimate this without communication cost as the difference of the last two models broadcasted by the server [43]:

$$\xi = \frac{w_{t_0} - w_{t_{-1}}}{\left\|w_{t_0} - w_{t_{-1}}\right\|_2}$$

## 3.8 Sketch Approximation

An optimal estimator for the square of the euclidean norm of $\overline{\Delta}_t$ can be achieved through the utilization of AMS sketches, as detailed in [44]. We will commence by outlining the fundamental characteristics and properties of AMS sketches.

**Theorem 5 (Sketch Properties)** *An AMS sketch of a vector $v \in \mathbb{R}^d$ is an $l \times m$ real matrix $\Xi$:*

$$\mathrm{sk}(v) = \Xi = \begin{bmatrix} \xi_1 & \xi_2 & \dots & \xi_l \end{bmatrix}^\top \in \mathbb{R}^{l \times m} \ , \text{ where } l \cdot m \ll d$$

*The following properties hold for AMS sketches:*

*(i) Complexity: The $l \times m$ sketch of a vector $v \in \mathbb{R}^d$ can be computed in $\mathcal{O}(ld)$ steps.*

*(ii) Linearity: Operator $\mathrm{sk}(\cdot)$ is linear.*

*(iii) Estimation: The function $\mathcal{M}_2(\Xi) = \underset{i=1,\dots,l}{\mathrm{median}} \|\xi_i\|_2^2$ is an excellent estimator of the euclidean norm of $v$ (within relative $\epsilon$-error):*

$$(1-\epsilon)\|v\|_2^2 \leq \mathcal{M}_2(\Xi) \leq (1+\epsilon)\|v\|_2^2 \quad \text{with probability at least } (1-\delta)$$

*where $l = \mathcal{O}(\log 1/\delta)$ and $m = \mathcal{O}\left(1/\epsilon^2\right)$.*

**Theorem 6 (Sketch FDA)** *Define $S_k(t) = \left[\left\|\Delta_t^{(k)}\right\|_2^2 \ \ \mathrm{sk}\left(\Delta_t^{(k)}\right)\right]^\top \in \mathbb{R}^{1+l\times m}$ where $l = \mathcal{O}(\log 1/\delta)$ and $m = \mathcal{O}\left(1/\epsilon^2\right)$. Also, define $F(v,\Xi) = v - \frac{1}{1+\epsilon}\mathcal{M}_2(\Xi)$. Then, the condition $F(S(t)) \leq \Theta$ implies the RTC with probability at least $(1-\delta)$.*

**Proof:**

$$F(S(t)) \overset{(3.5)}{=} F\left(\sum_{k=1}^K \frac{n_k}{n} S_k(t)\right) = \left(\sum_{k=1}^K \frac{n_k}{n}\left\|\Delta_t^{(k)}\right\|_2^2\right) - \frac{1}{1+\epsilon}\mathcal{M}_2\left(\sum_{k=1}^K \frac{n_k}{n}\mathrm{sk}\left(\Delta_t^{(k)}\right)\right) =$$

$$\overset{Linearity.(ii)}{=} \left(\sum_{k=1}^K \frac{n_k}{n}\left\|\Delta_t^{(k)}\right\|_2^2\right) - \frac{1}{1+\epsilon}\mathcal{M}_2\left(\mathrm{sk}\left(\sum_{k=1}^K \frac{n_k}{n}\Delta_t^{(k)}\right)\right) =$$

$$= \left(\sum_{k=1}^K \frac{n_k}{n}\left\|\Delta_t^{(k)}\right\|_2^2\right) - \frac{1}{1+\epsilon}\mathcal{M}_2\left(\mathrm{sk}\left(\overline{\Delta}_t\right)\right) \geq$$

$$\overset{(3.11)}{\geq} \left(\sum_{k=1}^K \frac{n_k}{n}\left\|\Delta_t^{(k)}\right\|_2^2\right) - \left\|\overline{\Delta}_t\right\|_2^2 \quad \text{with probability at least } (1-\delta) \tag{3.10}$$

$$\frac{1}{1+\epsilon}\mathcal{M}_2\left(\mathrm{sk}\left(\overline{\Delta}_t\right)\right) \overset{\text{Estimation.(iii)}}{\leq} \left\|\overline{\Delta}_t\right\|_2^2 \quad \text{with probability at least } (1-\delta) \tag{3.11}$$

We proved in (3.10) that $F(S(t))$ overestimates the actual variance (Theorem 1) with probability at least $(1-\delta)$. Hence, the condition $F(S(t)) \leq \Theta$ implies the RTC with probability at least $(1-\delta)$, completing the proof [43].

## 3.9 Synchronous strategy

The *synchronous*[2] strategy serves as a comparative baseline. It can be understood as a special case of the FDA algorithm (Algorithm 5) where $\Theta$ is set to zero[3]. Its structure harkens back to traditional non-dynamic Federated Learning training algorithms.

[2]While the name "synchronous" is used for this strategy, it's worth noting that the previous methods discussed are also synchronous in nature. The distinction here is in its similarity to the "Synchronous SGD" (Section 2.3.3), rather than its operational synchronicity.

[3]This renders operations regarding local/global states redundant, but is mentioned for clarity.

# Chapter 4

# Experiments

In the Experiments section, we undertake an empirical investigation of the performance of the three FDA strategies delineated in the preceding sections: "naive" (Section 3.6), "linear" (Section 3.7), and "sketch" (Section 3.8) within the Federated Learning (FL) setting. Our goal is to discern the relative merits and potential shortcomings of each strategy not only in juxtaposition with each other but also in comparison to the baseline "synchronous" strategy (Section 3.9).

## 4.1 Simulator

We have developed a Python package tailored for simulating the FDA algorithm (Algorithm 5) in the Federated Learning (FL) setting with customizable hyper-parameters. Built atop the TensorFlow framework, our package is designed for adaptability, allowing the evaluation of diverse experiments on a variety of Convolutional Neural Networks (CNNs). Although we initially crafted two specific CNNs for our experiments, the package's architecture is receptive to the integration of other Neural Network designs in future studies.

Emphasizing efficient execution, the package is highly parallelized, capable of simultaneously managing a spectrum of tasks—from a single simulation to orchestrating dozens or even hundreds of simulations. The only limitation, in this case, tends to be the available computational resources. With a focus on deployment flexibility, our package seamlessly functions in both High-Performance Computing (HPC) environments and local clusters. Beyond native support for SLURM execution [45] and integration with "Dask" [46], the package automatically utilizes all available CPU and GPU resources, ensuring efficient computation regardless of the setup.

To further contextualize our experiments, we will first detail the architectures of the two primary CNN models utilized: "LeNet-5" and "AdvancedCNN".

### 4.1.1 LeNet-5

"LeNet-5", introduced by LeCun et al. in 1998 [47], is one of the earliest Convolutional Neural Network (CNN) structures. Its historical significance and relatively small number of parameters (specifically 61.706) make it an attractive model for various applications, particularly in digit recognition tasks.

The concise yet comprehensive design of "LeNet-5" makes it an ideal choice for our experiments, allowing us to explore the behavior of the FDA algorithm with a well-established model.

Figure 4.1 illustrates the architecture of "LeNet-5", comprising multiple convolutional and pooling layers followed by fully connected layers. The structure's simplicity and

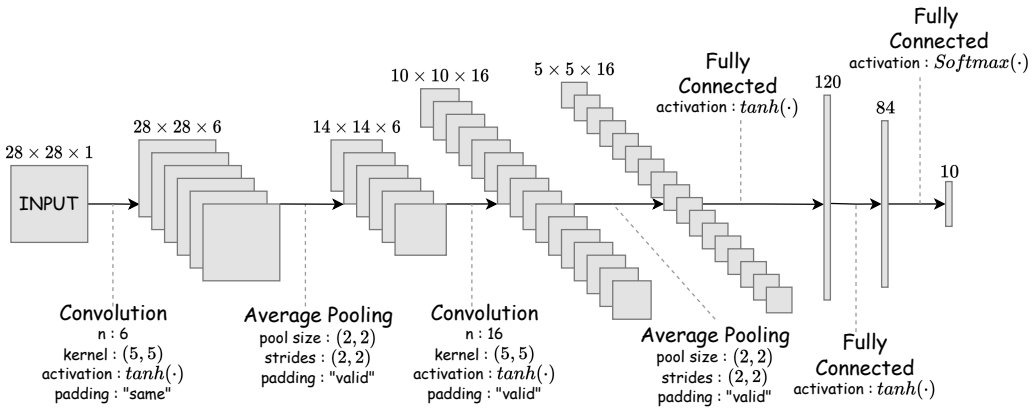



Figure 4.1: LeNet-5 architecture

efficiency have contributed to its enduring relevance in the field of deep learning.

### 4.1.2 AdvancedCNN

The "AdvancedCNN" is a custom-built Convolutional Neural Network (CNN) tailored for our experiments. Comprising 2.592.202 parameters, this architecture allows us to test the FDA algorithm in a more complex environment compared to LeNet-5. As depicted in Figure 4.2, the structure consists of multiple convolutional and pooling layers, followed by fully connected layers.

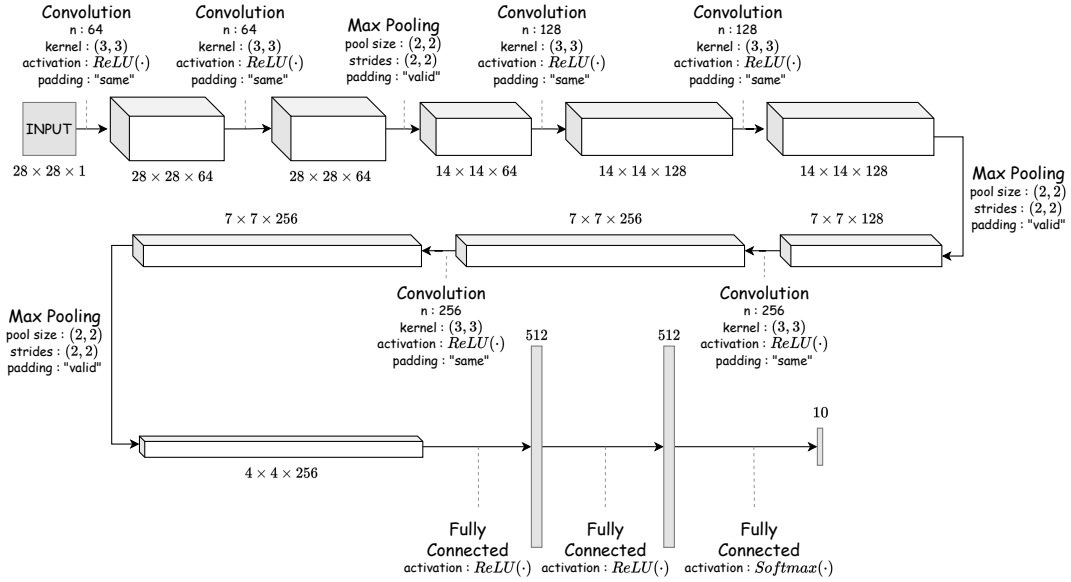



Figure 4.2: AdvancedCNN architecture

While the figure provides a detailed view of the structure, it's worth noting that two dropout layers are included in the architecture but not shown in the diagram. These dropout layers, with a rate of 0.5, are strategically placed after the dense layers to prevent overfitting.

The "AdvancedCNN" model offers a more intricate architecture, allowing us to investigate the FDA algorithm's performance with varying levels of complexity. The inclusion of dropout layers adds an additional layer of robustness to the model.

### 4.1.3 Training

Our experiments operate within the Federated Learning (FL) setting, leveraging the two Convolutional Neural Networks: "LeNet-5" (Section 4.1.1) and "AdvancedCNN" (Section 4.1.2). To initialize these networks, we employ random Xavier (Glorot uniform) initialization [48].

Training utilizes the MNIST dataset [49], which we partition into approximately equal segments for all participating clients, in alignment with the FL setting. In the context of the Functional Dynamic Averaging (FDA) algorithm (Algorithm 5), we test the four distinct techniques: "naive" (Section 3.6), "linear" (Section 3.7), "sketch" (Section 3.8) and the baseline "synchronous" (Section 3.9). All of our experimental designs align with the assumptions presented in Section 3.5. As a loss function we employ the cross-entropy, a gold standard for classification endeavors.

For evaluation, we introduce and leverage the term *epoch*. Within our experiments, an *epoch* refers to a single pass through the entire FL dataset, completed when all clients have trained once on their whole local dataset. The training for each local client uniformly adopts the Adam optimization method (Algorithm 2), maintaining parameters: $\eta = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$.

A fundamental constant in our varied simulations is the data allocation to each client, contingent, naturally, on the consistent number of clients. To illustrate, any simulation involving thirty clients will share an identical dataset across all its tests. Beyond this consistency, all other elements can be regarded as stochastic in nature: the shuffling of data, the Glorot uniform initialization, and more. This design choice is intentional. Given the extensive array of unique simulations, any variance from less ideal or highly favorable initializations or shuffles becomes inconsequential.

### 4.1.4 Hyper-Parameters

The Functional Dynamic Averaging (FDA) algorithm (Algorithm 5) necessitates the specification of several hyper-parameters. These include the number of clients ($K$), the local mini-batch size ($b$), the monitoring threshold ($\Theta$), the strategy that defines $S_k(t)$ and $F(\cdot)$, and the number of local steps ($s$) which is consistently set to one throughout our testing.

The hyper-parameters and their corresponding values are summarized in Table 4.1.

| **Parameter** | **Values** |
|---|---|
| CNN name | {"LeNet-5", "AdvancedCNN"} |
| FDA method | {"naive", "linear", "sketch", "synchronous"} |
| $K$ (Number of clients) | $\{5, 10, 15, \ldots, 60\}$ |
| $b$ (Local mini-batch size) | $\{32, 128, 256\}$ |
| $\Theta$ (Monitoring threshold) | $\{0.5, 1, 2\}$ |
| $E$ (Epochs, as defined in 4.1.3) | (Dynamic) |

Table 4.1: Hyper-parameters used in the experiments. [1]

In our experimental design, we define a *simulation* as the process of training in the FL setting using a specific strategy, guided by a particular combination of hyper-parameters as outlined in Table 4.1. The number of epochs for each simulation is predetermined, yet it dynamically varies depending on a combination of factors. These include the specific FDA strategy in use, the selected hyper-parameters like the monitoring threshold ($\Theta$), local mini-batch size ($b$) and the number of clients ($K$), as well as the choice of model

[1] For the baseline "synchronous" strategy (Section 3.9), the hyper-parameter $\Theta$ is not applicable and can be disregarded.

between "LeNet-5" and "AdvancedCNN". This adaptively chosen number of epochs serves as the predetermined stopping criterion for our experiments.

The selection of hyper-parameters was not arbitrary or made without consideration. Extensive exploratory testing was conducted to identify the most suitable values for investigation. This process, although vital to the experimental design, is beyond the scope of this work and will not be detailed further.

By conducting comprehensive testing across all possible combinations of the above hyper-parameters with both models and all four strategies, we were able to record a wealth of results and useful metrics.

### 4.1.5 Data Analysis

Our experiments unfold in a simulated environment, a departure from a real-world Federated Learning context. While we model all pertinent costs, such as communication and computation time, it's essential to understand these as approximations. For clarity, when transmitting a 32-bit float from clients to the server, we account for 4 bytes and vice versa. Additionally, the local computation time, which is largely driven by backpropagation, is empirically determined for each batch size, capturing the average time consumed, and this measurement is then applied in our analyses based on the specific batch size used.

While our analysis isn't precisely tailored to reflect real-world scenarios in their entirety, it is meticulously crafted to ensure valid and equitable comparisons between the different strategies.

For the computation of transmission times, we turn to the two communication models detailed in Section 2.5.2. Specifically, the transmission time for a data amount $C$ is calculated using equations (2.4) for the Common channel communication, and (2.5) for the Hypercube communication model. In both scenarios, we operate under the premise of a channel bandwidth $B$ set at 1Gbit.

In the subsequent Results section (Section 4.2), we grapple with the challenge of visualizing and comparing the performance of various strategies across a multitude of hyper-parameters. The extensive data generated from different simulations, each representing a unique combination of hyper-parameters, makes creating a comprehensive and meaningful comparison a complex task.

To aid this comparative analysis, we introduce the notion of an *Accuracy Target*. The essence is to juxtapose different strategies based on the epoch in which they first attain this accuracy target, assessing their efficiency through the lens of the unified time cost (Equation (2.3)).

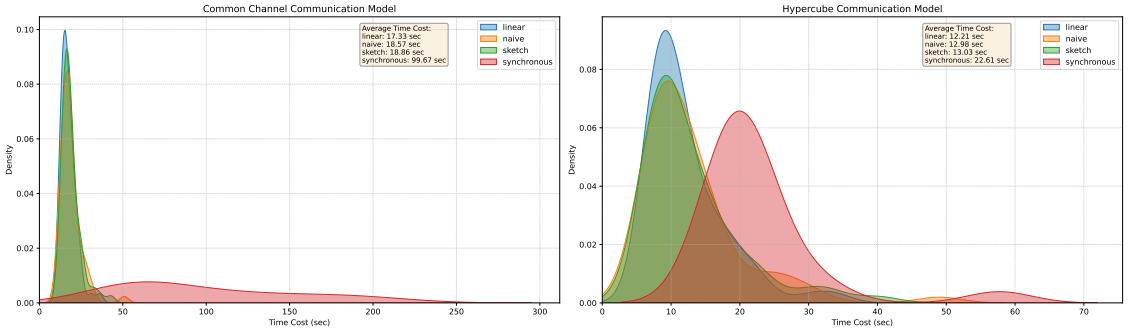
For illustration, imagine two simulations—one where the "sketch" strategy meets the accuracy target by the tenth epoch and another where the "synchronous" strategy reaches the same in merely the first epoch. Our comparison will focus on the specific epoch where the target is first met, rather than subsequent epochs where the target may be repeatedly achieved. It's worth noting that some simulations might not reach the accuracy target, and therefore, they will not be included at all.

We term this evaluative paradigm as the *Filtered* approach. It offers an unambiguous, equitable assessment, spotlighting the efficiency of strategies in accomplishing the set accuracy.

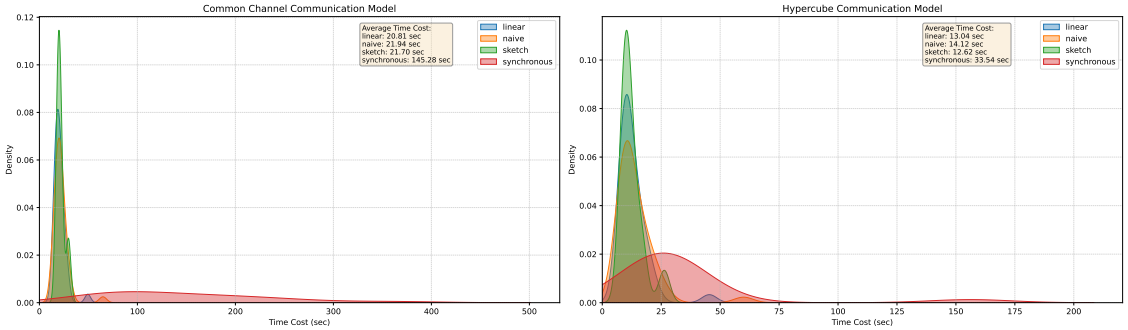
## 4.2 Results

In this section, we delve into the results, beginning with high-level overview plots that encapsulate the essence of all conducted simulations. Then, we'll transition to showcasing select examples from the comprehensive collection of plots available in Appendix A

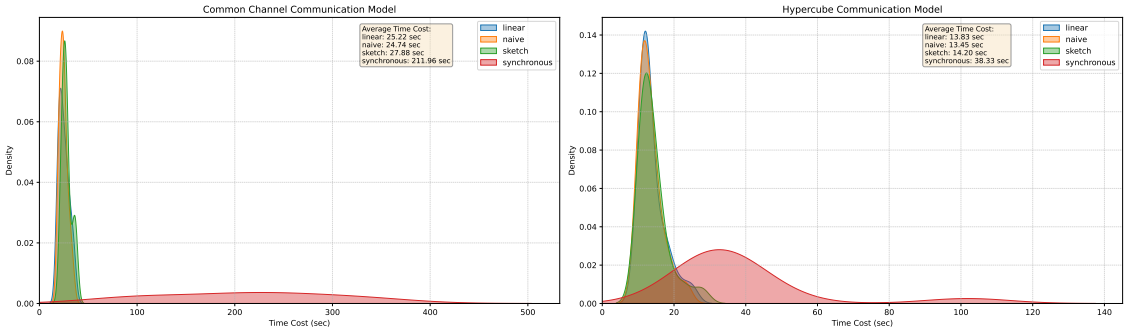
### 4.2.1 LeNet-5



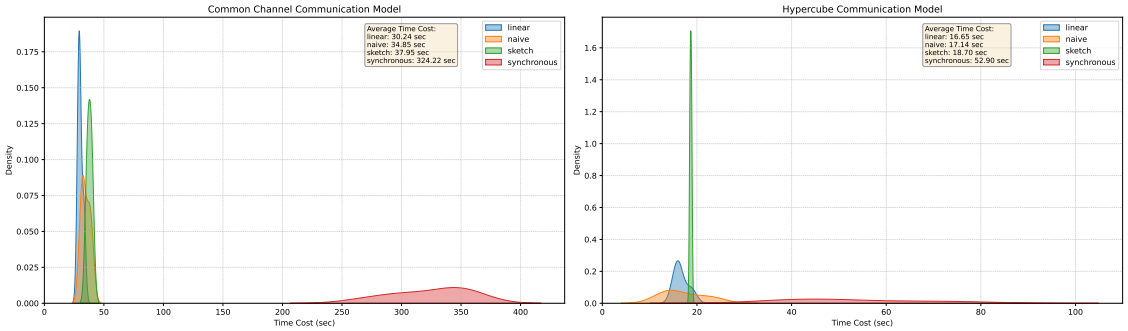

(a) Accuracy Target is 0.96.




(b) Accuracy Target is 0.965.

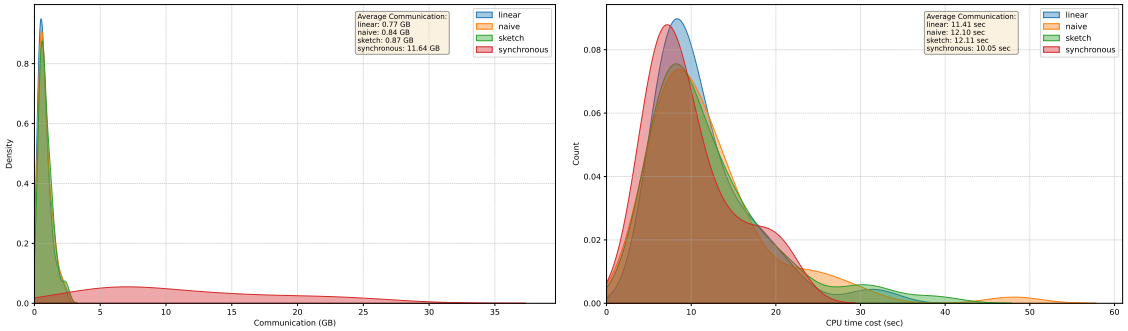



(c) Accuracy Target is 0.97.

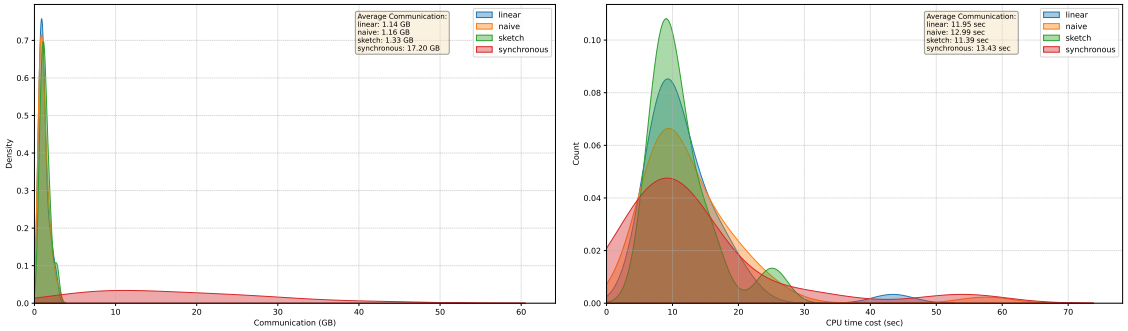



(d) Accuracy Target is 0.975.

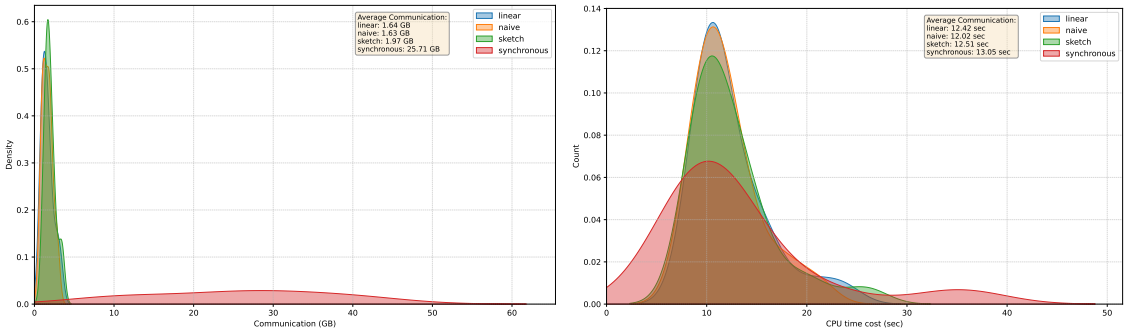
Figure 4.3: *Filtered* (Section 4.1.5). Kernel Density Estimation plots offering a high-level overview of the efficiency of different strategies. The density curves represent the distribution of time costs incurred – as defined in (2.3) – at the first epoch where the accuracy target is met. The height of the curve at any given point indicates the estimated density of simulations with that specific time cost.



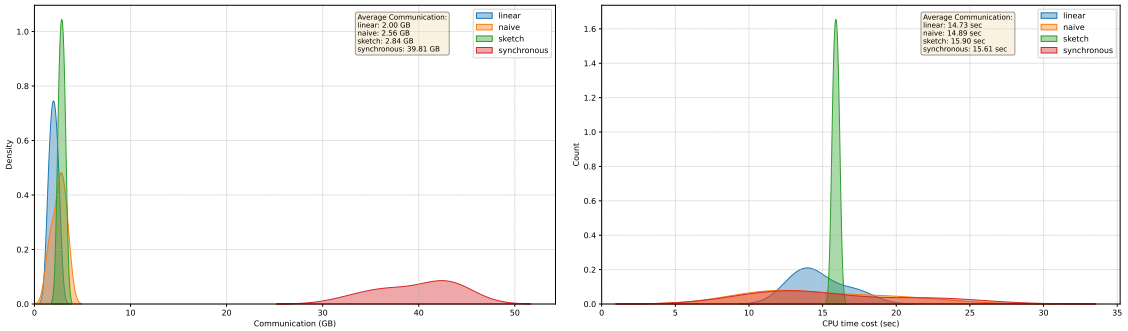

(a) Accuracy Target is 0.96.




(b) Accuracy Target is 0.965.




(c) Accuracy Target is 0.97.




(d) Accuracy Target is 0.975.

Figure 4.4: *Filtered* (Section 4.1.5). Kernel Density Estimation plots offering a high-level overview of the efficiency of various strategies. The density curves represent the distribution of communication costs and CPU time costs incurred at the first epoch where the accuracy target is met. The height of the curves at any given point indicate the estimated density of simulations with that specific cost.

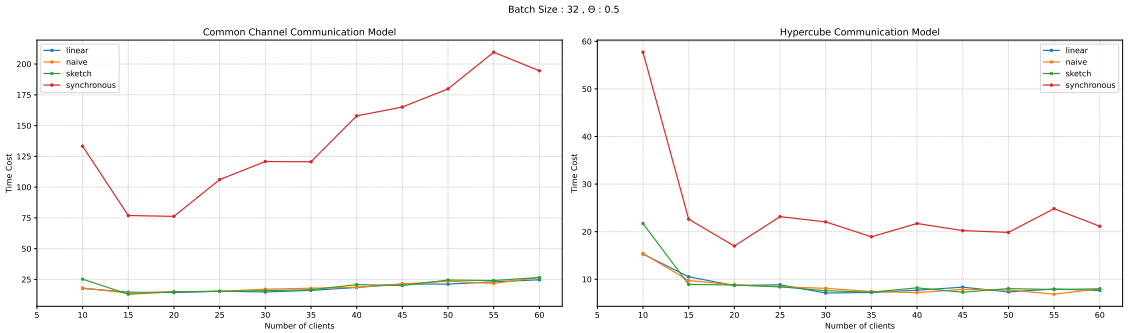



Figure 4.5: *Filtered* (Section 4.1.5). Detailed view of individual simulations with a batch size ($b$) of 32 and a monitoring threshold ($\Theta$) set to 0.5. We illustrate the time cost incurred at the first epoch where the accuracy target of 0.96 is met. Each dot within a chain of line segments represents an individual simulation. For the exhaustive collection of results, refer to Appendix A.A.

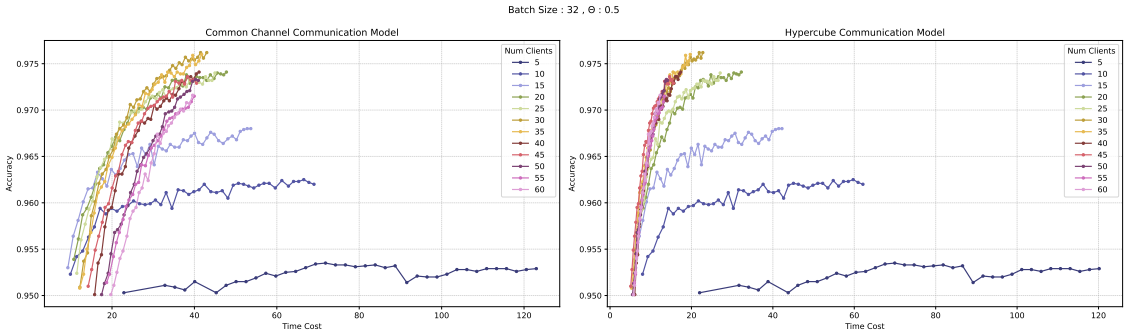
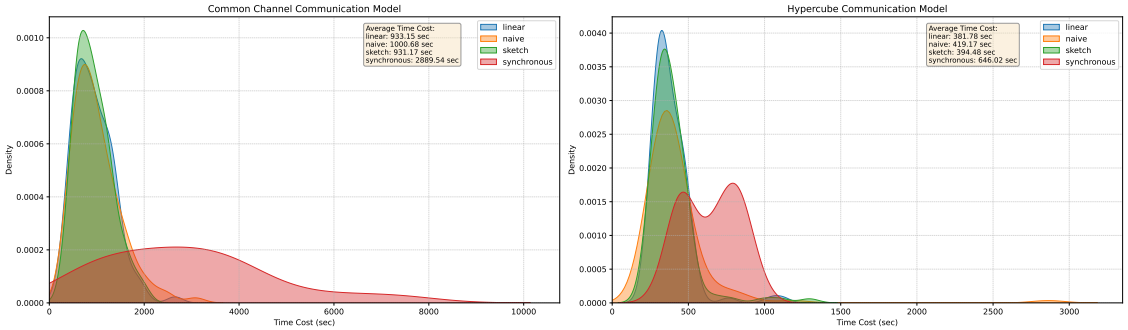
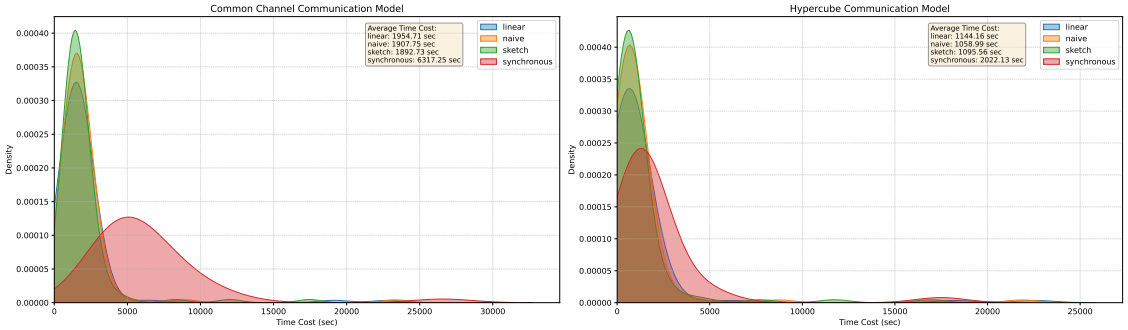



Figure 4.6: Detailed view an individual simulation's progression in time, using the "sketch" strategy, with a batch size ($b$) of 32 and a monitoring threshold ($\Theta$) set to 0.5, after reaching the accuracy target of 0.95. Each chain of line segments represents an individual simulation. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment. For the exhaustive collection of results, refer to Appendix A.A.

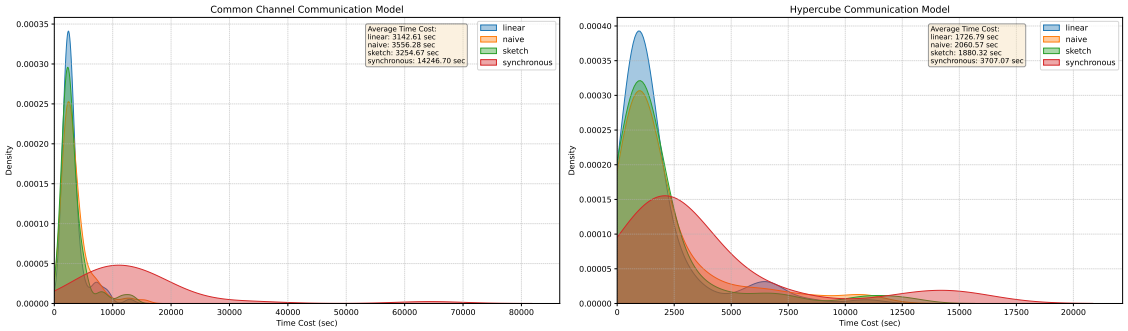
### 4.2.2 AdvancedCNN



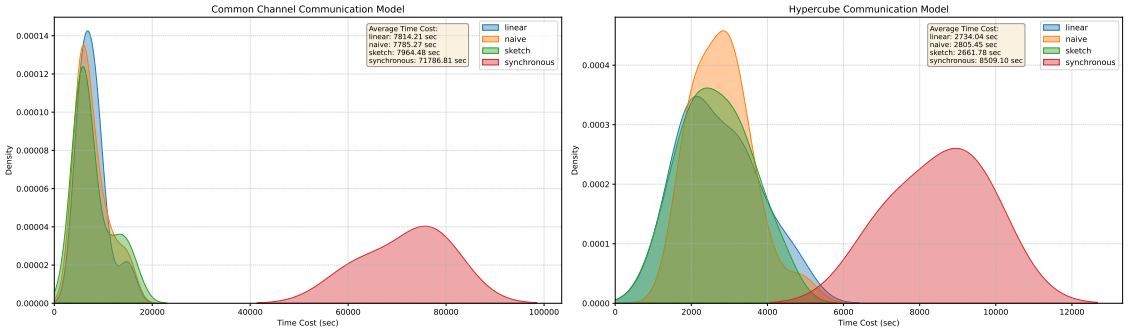

(a) Accuracy Target is 0.97.




(b) Accuracy Target is 0.98.

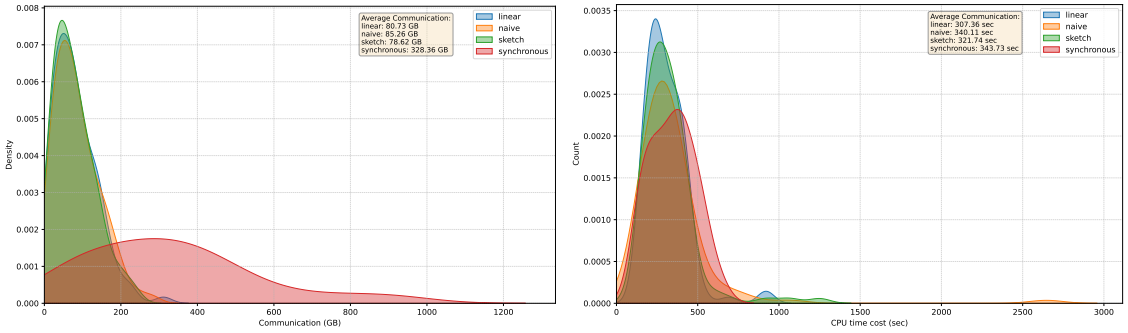



(c) Accuracy Target is 0.985.

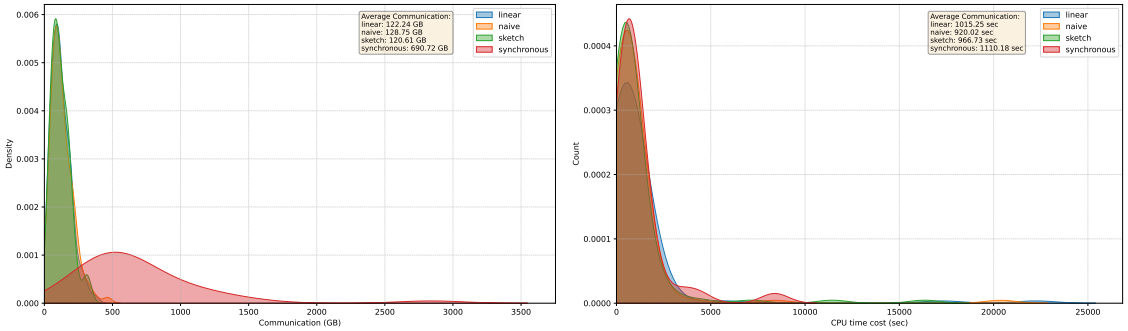



(d) Accuracy Target is 0.99.

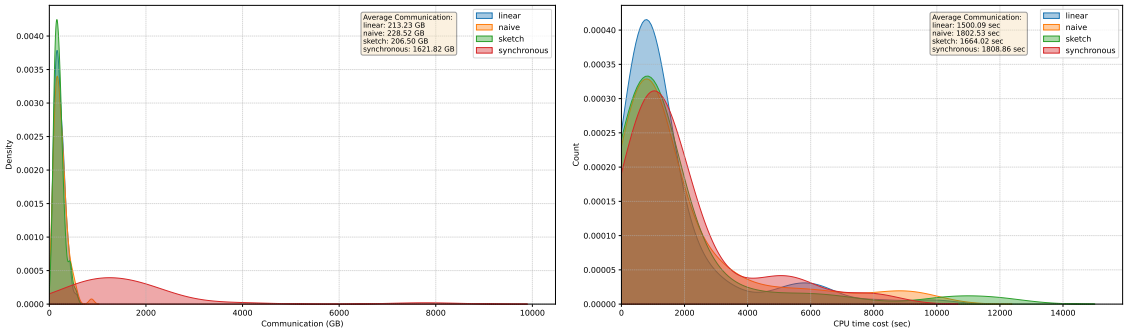
Figure 4.7: *Filtered* (Section 4.1.5). Kernel Density Estimation plots offering a high-level overview of the efficiency of different strategies. The density curves represent the distribution of time costs incurred – as defined in (2.3) – at the first epoch where the accuracy target is met. The height of the curve at any given point indicates the estimated density of simulations with that specific time cost.



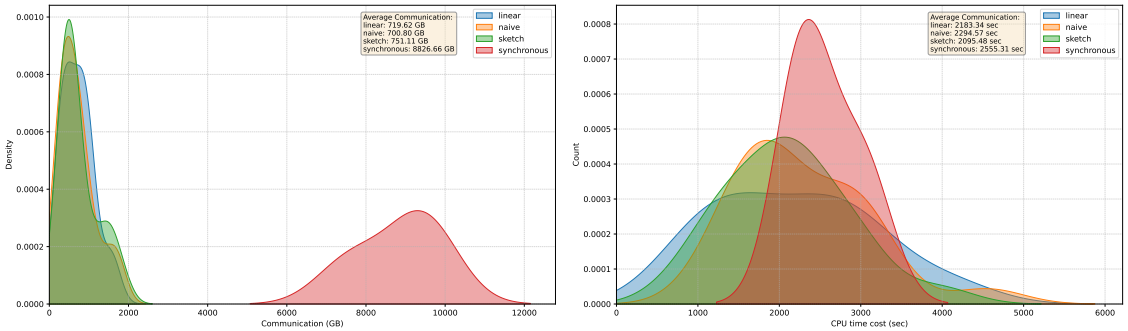

(a) Accuracy Target is 0.97.




(b) Accuracy Target is 0.98.




(c) Accuracy Target is 0.985.




(d) Accuracy Target is 0.99.

Figure 4.8: *Filtered* (Section 4.1.5). Kernel Density Estimation plots offering a high-level overview of the efficiency of various strategies. The density curves represent the distribution of communication costs and CPU time costs incurred at the first epoch where the accuracy target is met. The height of the curves at any given point indicate the estimated density of simulations with that specific cost.

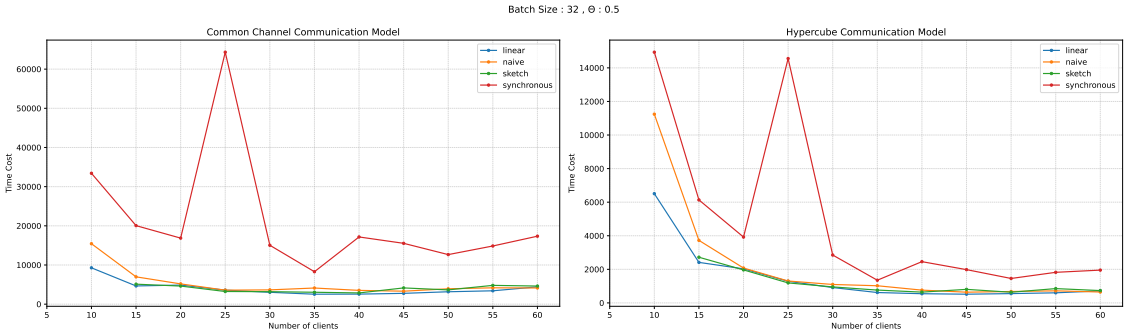



Figure 4.9: *Filtered* (Section 4.1.5). Detailed view of individual simulations with a batch size ($b$) of 32 and a monitoring threshold ($\Theta$) set to 0.5. We illustrate the time cost incurred at the first epoch where the accuracy target of 0.985 is met. Each dot within a chain of line segments represents an individual simulation. For the exhaustive collection of results, refer to Appendix A.B.

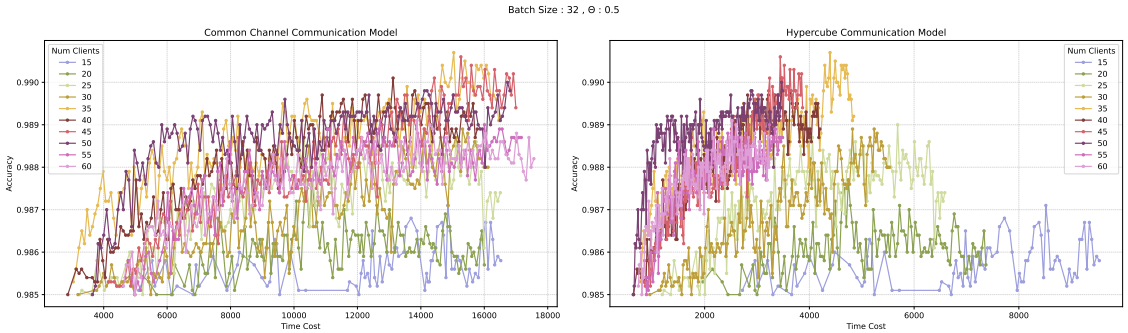



Figure 4.10: Detailed view an individual simulation's progression in time, using the "sketch" strategy, with a batch size ($b$) of 32 and a monitoring threshold ($\Theta$) set to 0.5, after reaching the accuracy target of 0.985. Each chain of line segments represents an individual simulation. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment. For the exhaustive collection of results, refer to Appendix A.B.

### 4.2.3 Discussion

In the previous section, we presented several density plots capturing the time cost incurred at the first epoch where different accuracy targets are achieved by individual simulations (Figures 4.3 and 4.7). Given the multitude of experiments, specifying hyper-parameter details were omitted for clarity. Instead, we emphasized on the overall performance trends across different strategies, including the baseline algorithm. Within each plot, we also provided information indicating the average cost for each strategy, offering a summarized view of their representative performance.

We begin by evaluating the performance of different strategies using the relatively small "LeNet-5" model. As depicted in Figure 4.3, all three FDA strategies exhibit similar performance across all accuracy targets. Consequently, we group them together for a collective comparison against the "synchronous" strategy, which follows a non-dynamic, predetermined round termination typical of many Federated Learning algorithms. Table 4.2 encapsulates the improvement achieved by employing FDA strategies over the traditional "synchronous" approach.

| | Accuracy Target | | | |
|---|---|---|---|---|
| | **0.96** | **0.965** | **0.97** | **0.975** |
| Common Channel Communication Model | 5.5 | 6.8 | 8.2 | 9.4 |
| Hypercube Communication Model | 1.8 | 2.5 | 2.8 | 3 |

Table 4.2: "LeNet-5": Comparative Speedup of FDA Strategies vs. "Synchronous" Strategy

The observed speedup in our experiments is not only substantial but also exhibits a pattern where the improvement correlates with the accuracy target. This behavior aligns with our expectations, as discussed in Section 3.2. During the early stages of training, round terminations occur frequently as clients diverge more easily, leading the FDA strategies to prescribe regular round terminations. As training progresses and the models approach higher quality, the impact of the FDA strategies becomes more pronounced. When training for the highest achievable accuracy, known to be approximately 0.98 for "LeNet-5", we observe a speedup of 11 for the Common Channel Communication model and 3 for the Hypercube communication model. These results hold particular promise in the context of neural networks, where the phenomenon of "diminishing returns" is well-known. As a model nears its learning limits for a given data-set and architecture, each additional increment in accuracy may necessitate a disproportionate increase in training time, tuning, and resources. The FDA strategies, growing more effective as training advances, can substantially enhance the accuracy limits of models in the FL setting, where the classic non-dynamic approach might otherwise stumble.

To elucidate how the "speedup" numbers were derived, consider the example of an accuracy target equal to 0.96 using the Common Channel Communication model. The calculation, derived from the informative text about the averages in Figure 4.3a, is given by $\frac{99.67}{(17.33+18.57+18.86)/3} \approx 5.5$.

Continuing our exploration, we shift focus to the "AdvancedCNN" model. As with the "LeNet-5", Figure 4.7 demonstrates that the FDA strategies maintain consistent performance across all accuracy targets. Given this consistency, we once again group them for a direct comparison with the "synchronous" approach. Table 4.3 encapsulates the improvement achieved by employing FDA strategies over the traditional "synchronous" strategy.

For both the "LeNet-5" and the "AdvancedCNN" models, our experiments revealed a consistent trend: the advantage of FDA strategies generally amplifies as the accuracy target escalates. Training for the highest achievable accuracy, which is roughly 0.99 for the

| | Accuracy Target | | | |
|---|---|---|---|---|
| | **0.97** | **0.98** | **0.985** | **0.99** |
| Common Channel Communication Model | 3 | 3.3 | 4.3 | 9.2 |
| Hypercube Communication Model | 1.6 | 1.8 | 2 | 3.1 |

Table 4.3: "AdvancedCNN": Comparative Speedup of FDA Strategies vs. "Synchronous" Strategy

"AdvancedCNN", resulted in a speedup of 11 with the Common Channel Communication model and 3 with the Hypercube communication model.

Upon examination of Figures 4.4 and 4.8, a counter-intuitive finding arises. In Federated Learning, it's commonly understood that fewer round terminations typically translate to longer convergence times. However, against this prevalent expectation, the FDA methods show nearly identical CPU time distributions when compared to the "synchronous" approach for achieving the same accuracy level. This suggests that while FDA methods have fewer synchronizations, they may be offering a more efficient exploration of the parameter space. Conversely, the regular realignments in the "synchronous" strategy might confine exploration, limiting clients due to constant synchronizations and potentially hampering their ability to locate optimal parameters effectively.

Building on the insights from Figures 4.4 and 4.8, we further detail the raw communication costs required to attain the designated accuracy targets. Regardless of the methodology for correlating communication costs with time expenditure (as detailed in Section 2.5.2), the raw communication data is undeniable. Considering the similar local computation times and the clear results from raw communication costs, the benefits of employing the FDA methods in Federated Learning are evident.

Turning our attention to Figures 4.5 and 4.9, we can observe the trends of simulations with specific batch size ($b$) and monitoring threshold ($\Theta$). Across the board, the three FDA strategies showcase consistent performance. These figures provide a snapshot; they only hint at the comprehensive collection of results cataloged in Appendix A. Within the Common Channel model, there emerges a distinct trend: the cost-effectiveness of achieving a particular accuracy target largely remains unaffected by the selection of clients for the simulation. Conversely, the Hypercube communication model, due to its efficient communication capabilities, seems to favor deploying a greater number of clients – a trend reflected in the time cost assessments.

Lastly, in Figures 4.6 and 4.10, we delve deeper, showcasing the progression of individual simulations in time after meeting a specific accuracy target. While these detailed plots may seem complex at first glance, they effectively highlight the intricacies and challenges of achieving convergence in the Federated Learning (FL) setting. Conclusive insights from these plots are best derived by examining the comprehensive collection of results in Appendix A. By offering this granular view, we aim to emphasize the depth of training involved and the challenges faced in the FL setting.

# Chapter 5

# Conclusion

The Functional Dynamic Averaging (FDA) offers a renewed approach to the traditional Federated Learning (FL) training methodology, inspired by the principles of Functional Geometric Monitoring (FGM). The central aim of this work was to explore the capabilities of the FDA approximation algorithm in the FL setting. Through experimental evaluations on two distinct CNNs, "LeNet-5" and "AdvancedCNN", it was evident that the FDA strategies consistently outperformed a traditional non-dynamic FL training method, reminiscent of FedSGD with small local steps. The FDA approximation algorithm, in essence, serves as a naive embodiment of the FGM protocol. Its demonstrated effectiveness not only stands on its own merit but also suggests FGM's potential to refine and enhance the conventional FL iterative training process into a more agile, adaptable, and efficient paradigm.

Building on this foundation, the next logical step is to delve deeper into the implications of the FDA algorithm in diverse contexts. One such avenue for future exploration emerges from our foundational assumption: all clients participate in every FL round. Analyzing the dynamics of various client selection strategies, both empirically and theoretically, will undoubtedly provide insights into FDA's performance and adaptability in diverse FL scenarios.

An additional layer of complexity is introduced when we consider the choice of optimizer. In our work, we anchored the experiments on the Adam optimizer for local client training. Yet, the relationship between FDA and a spectrum of optimizers, from SGD to AdaGrad, remains uncharted territory. Exploring these interactions could further highlight FDA's versatility across different FL environments.

Lastly, an intriguing dimension worth pursuing stems from the choice to monitor the variance of the models, a decision grounded in a wealth of empirical evidence. The potential of alternatives, such as cosine similarity, could provide valuable insights and pave the way for future research directions in refining the traditional Federated Learning iterative training process.

# Appendices

# Appendix A

# Detailed Results

## A.A LeNet-5

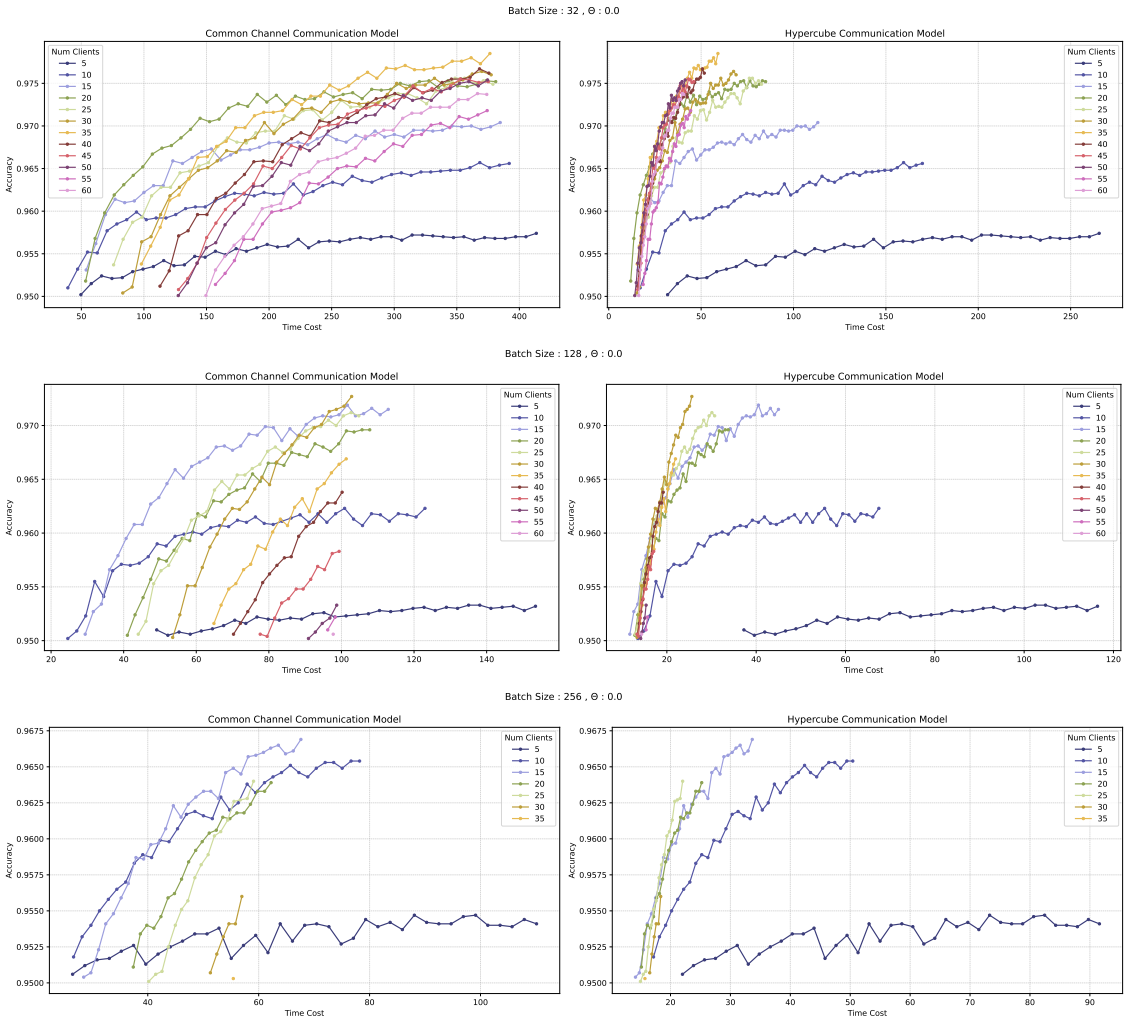



Figure A.1: Synchronous Strategy. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.95. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

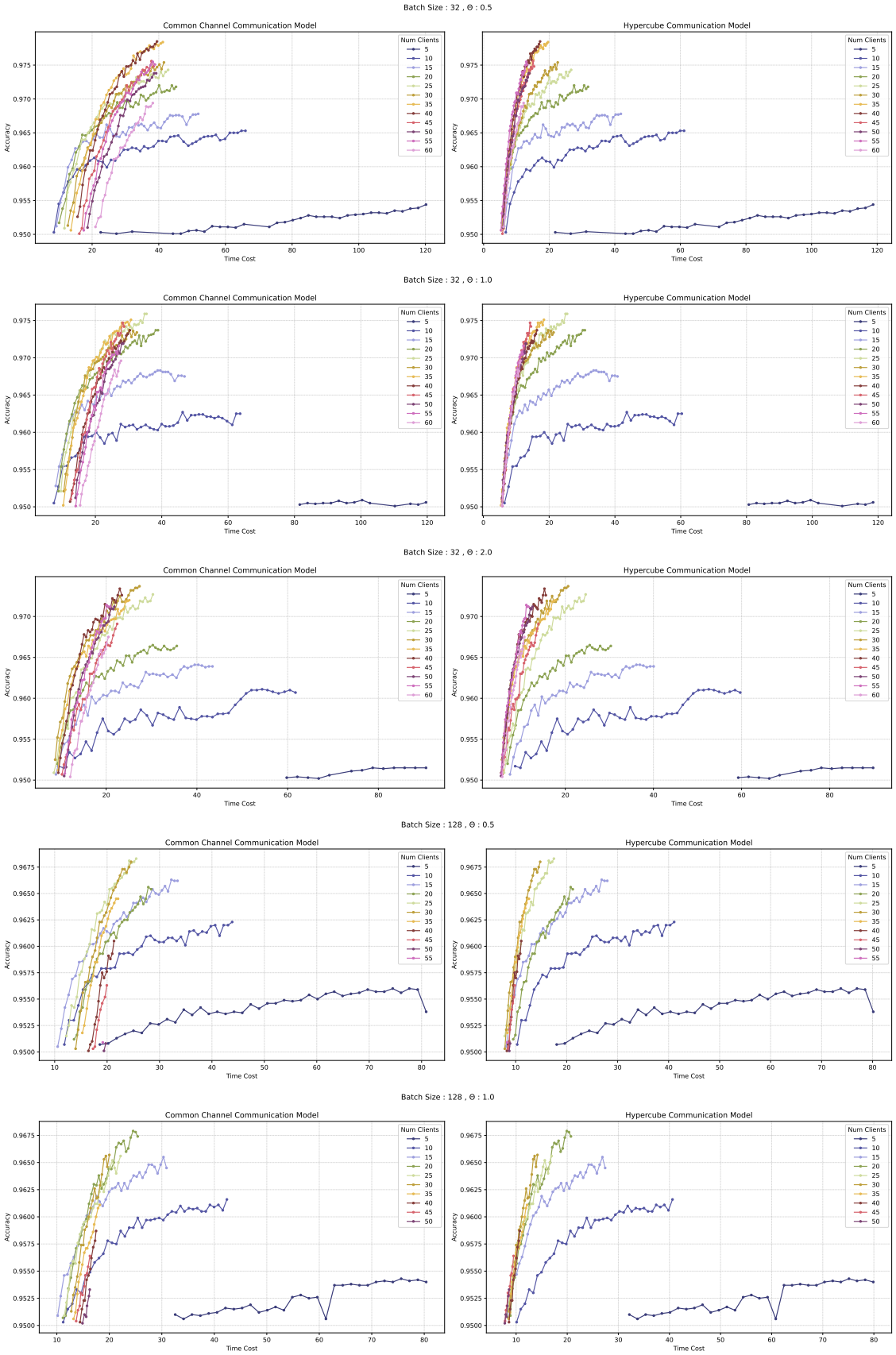



Figure A.2: Naive FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.95. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

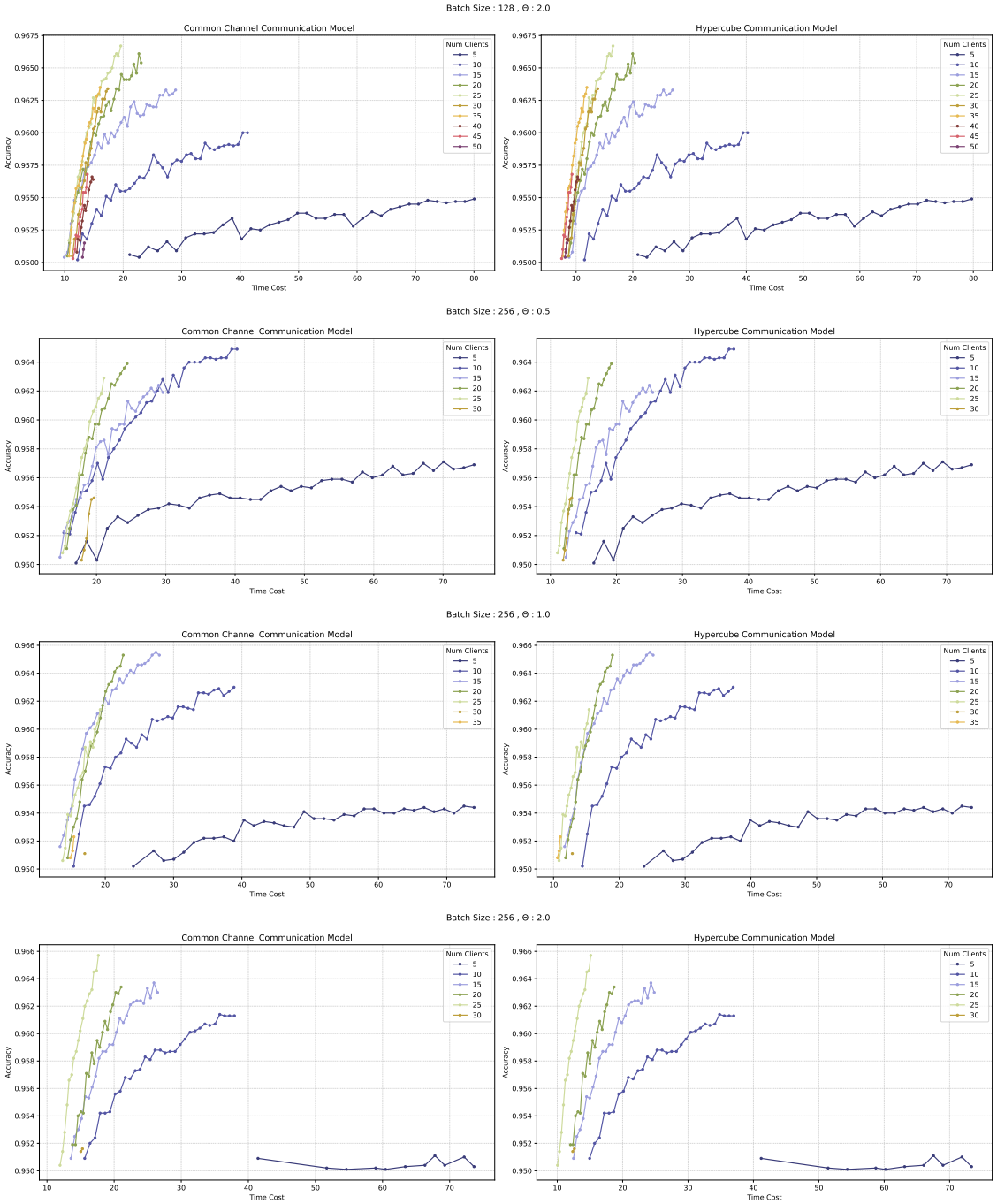



Figure A.2: Naive FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.95. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

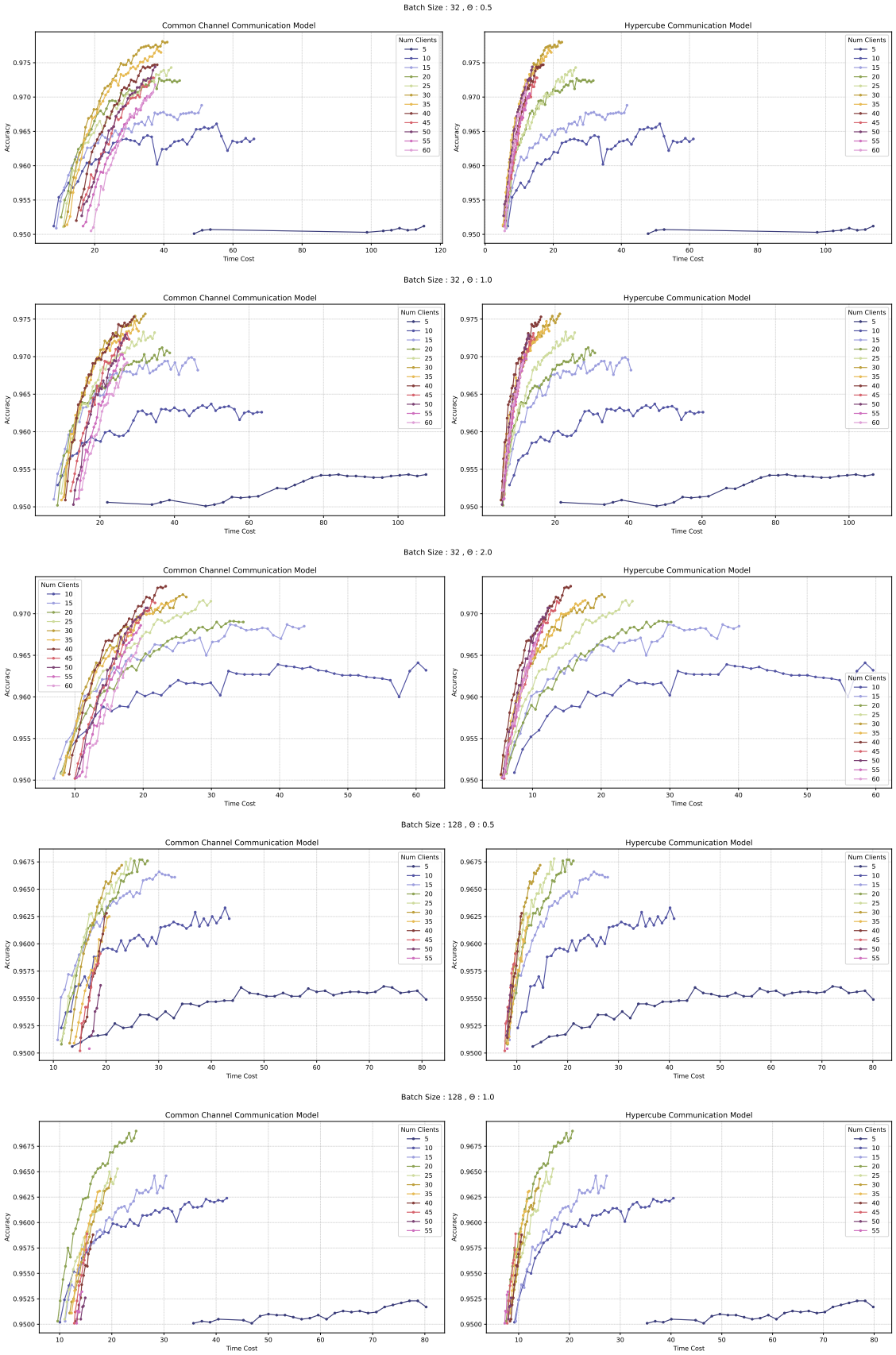



Figure A.3: Linear FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.95. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

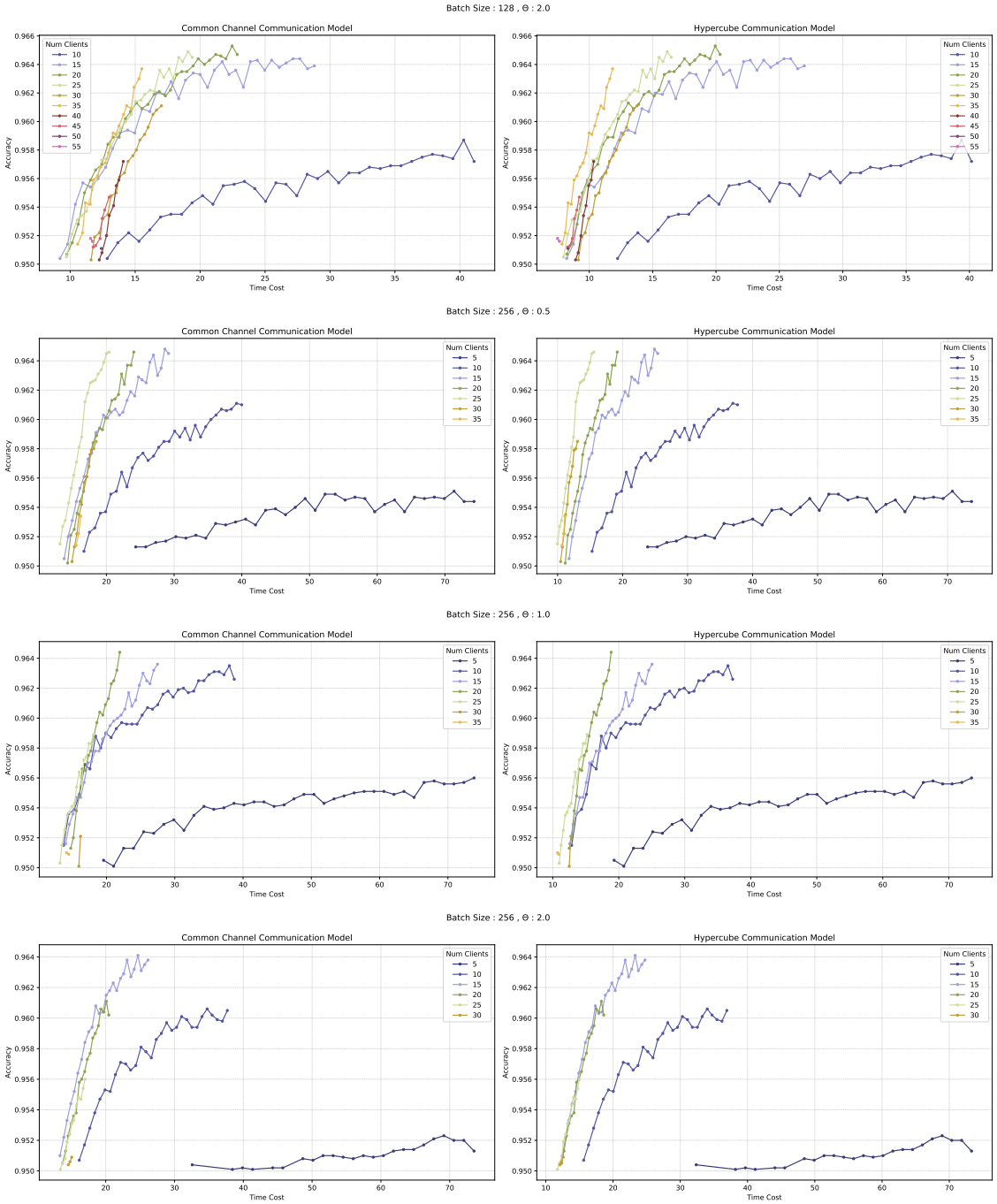



Figure A.3: Linear FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.95. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

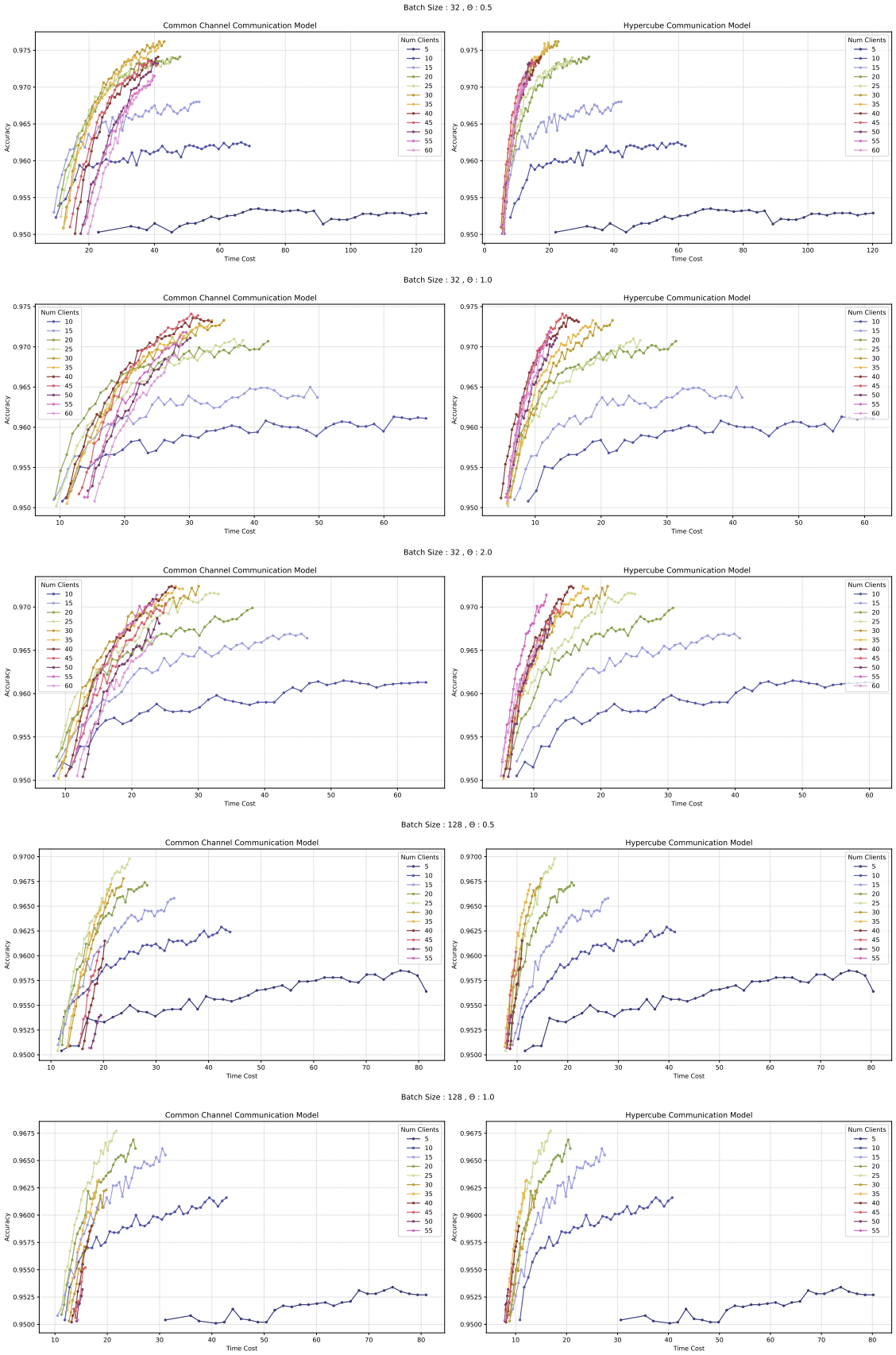



Figure A.4: Sketch FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.95. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

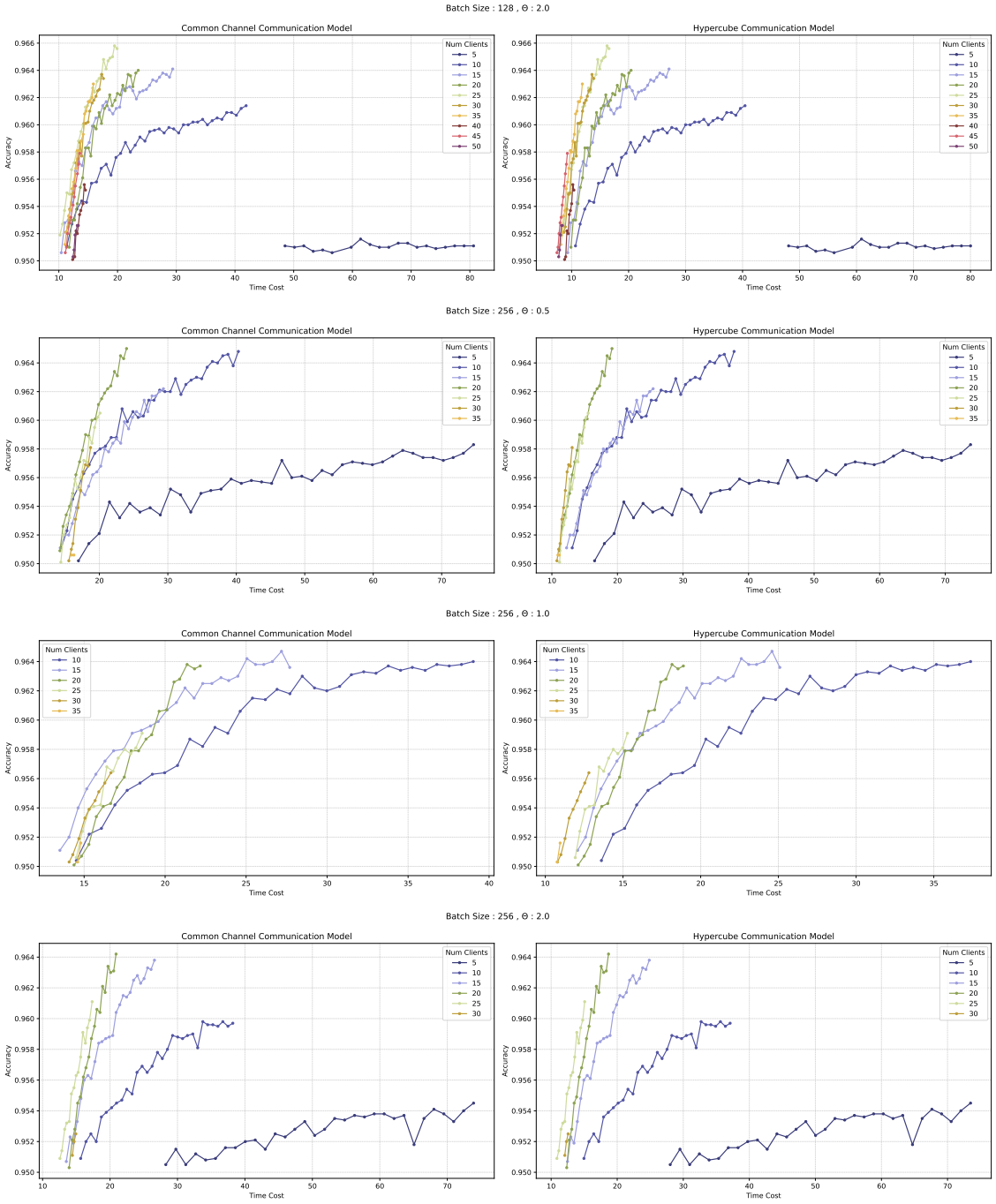



Figure A.4: Sketch FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.95. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

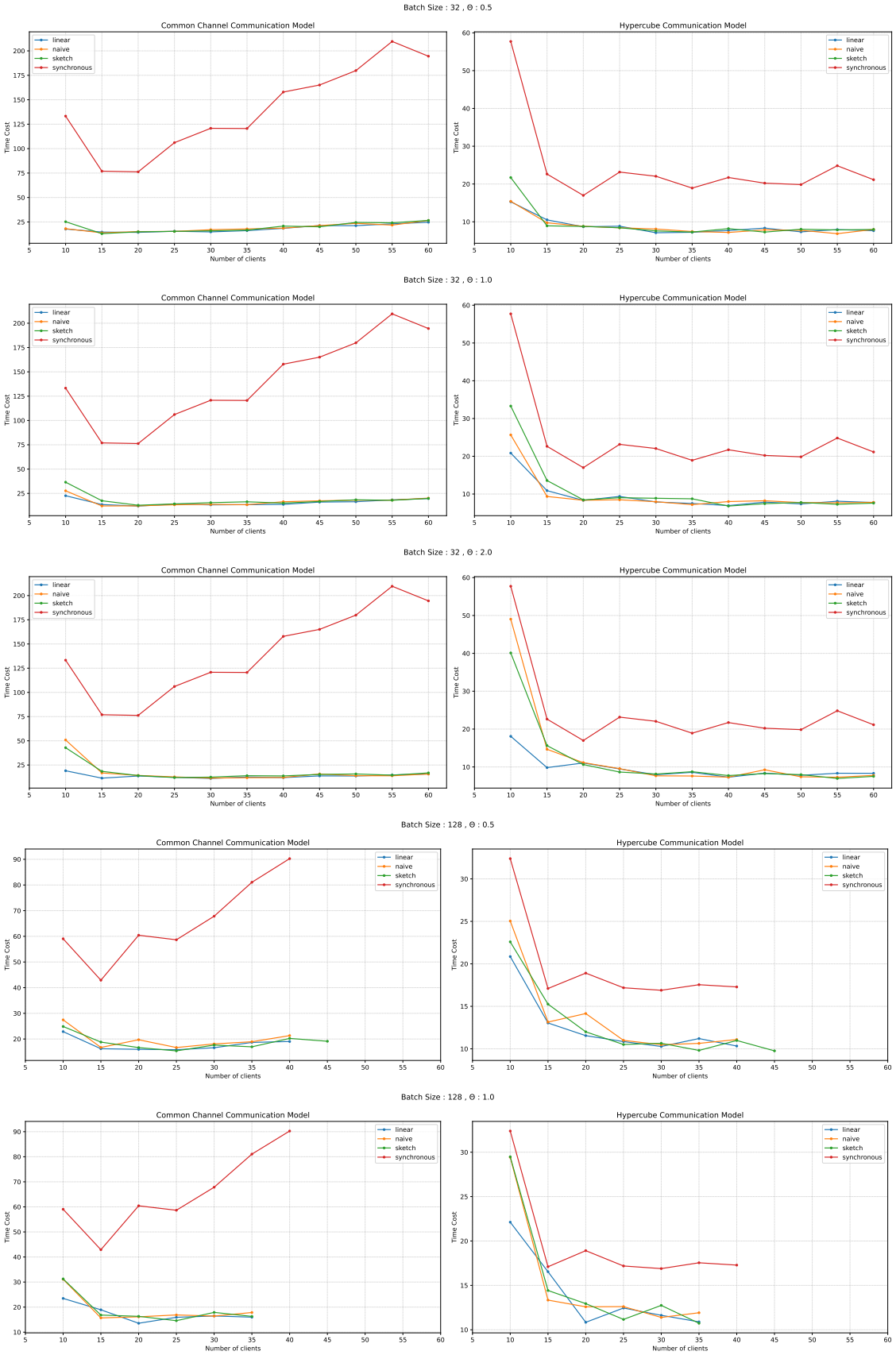



Figure A.5: *Filtered* (Section 4.1.5). Each dot within a chain of line segments represents an individual simulation, capturing the time cost incurred after reaching the accuracy threshold of 0.96. We illustrate the trends of the various methods in respect to the number of clients ($K$).

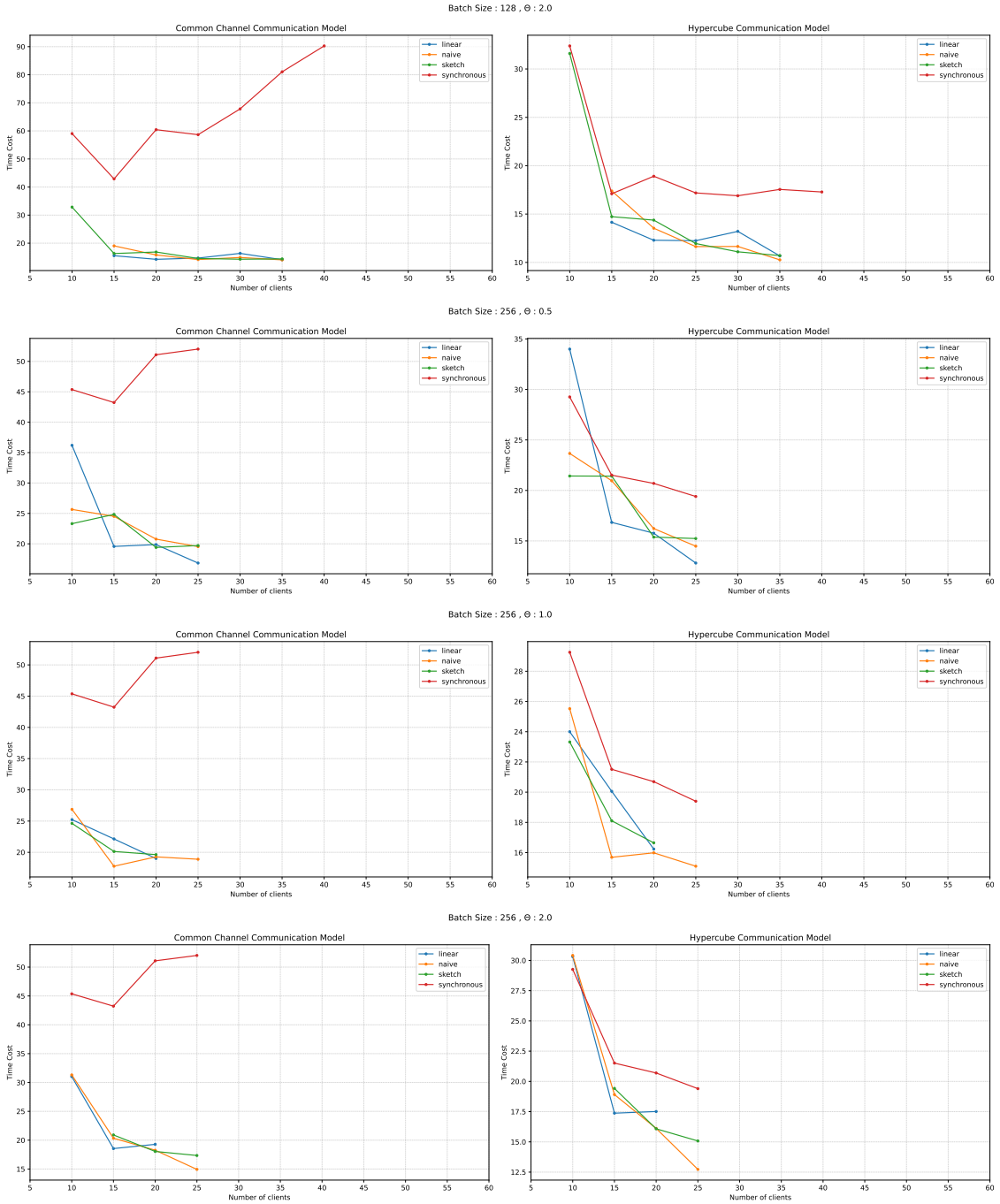



Figure A.5: *Filtered* (Section 4.1.5). Each dot within a chain of line segments represents an individual simulation, capturing the time cost incurred after reaching the accuracy threshold of 0.96. We illustrate the trends of the various methods in respect to the number of clients ($K$).

## A.B AdvancedCNN

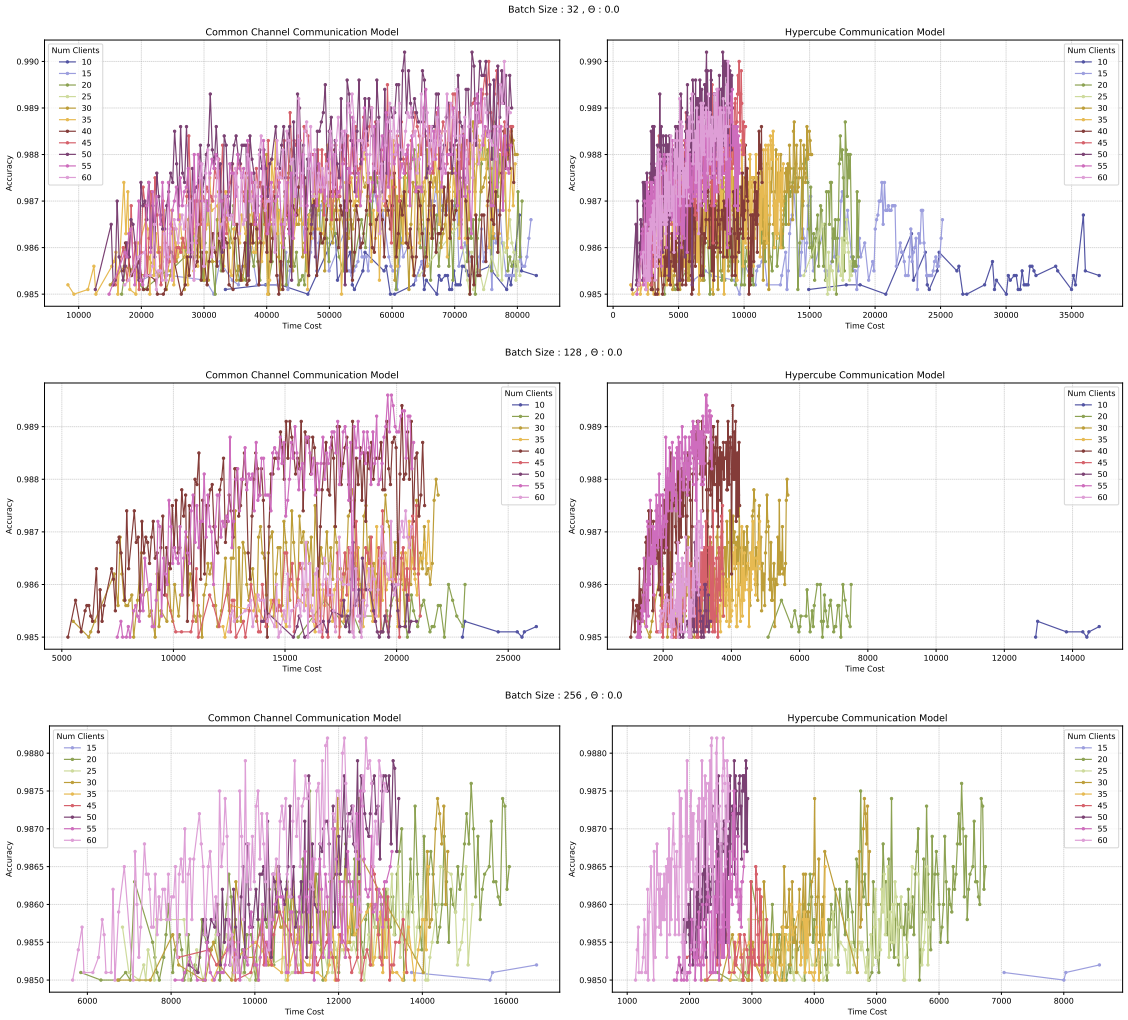



Figure A.6: Synchronous Strategy. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.985. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

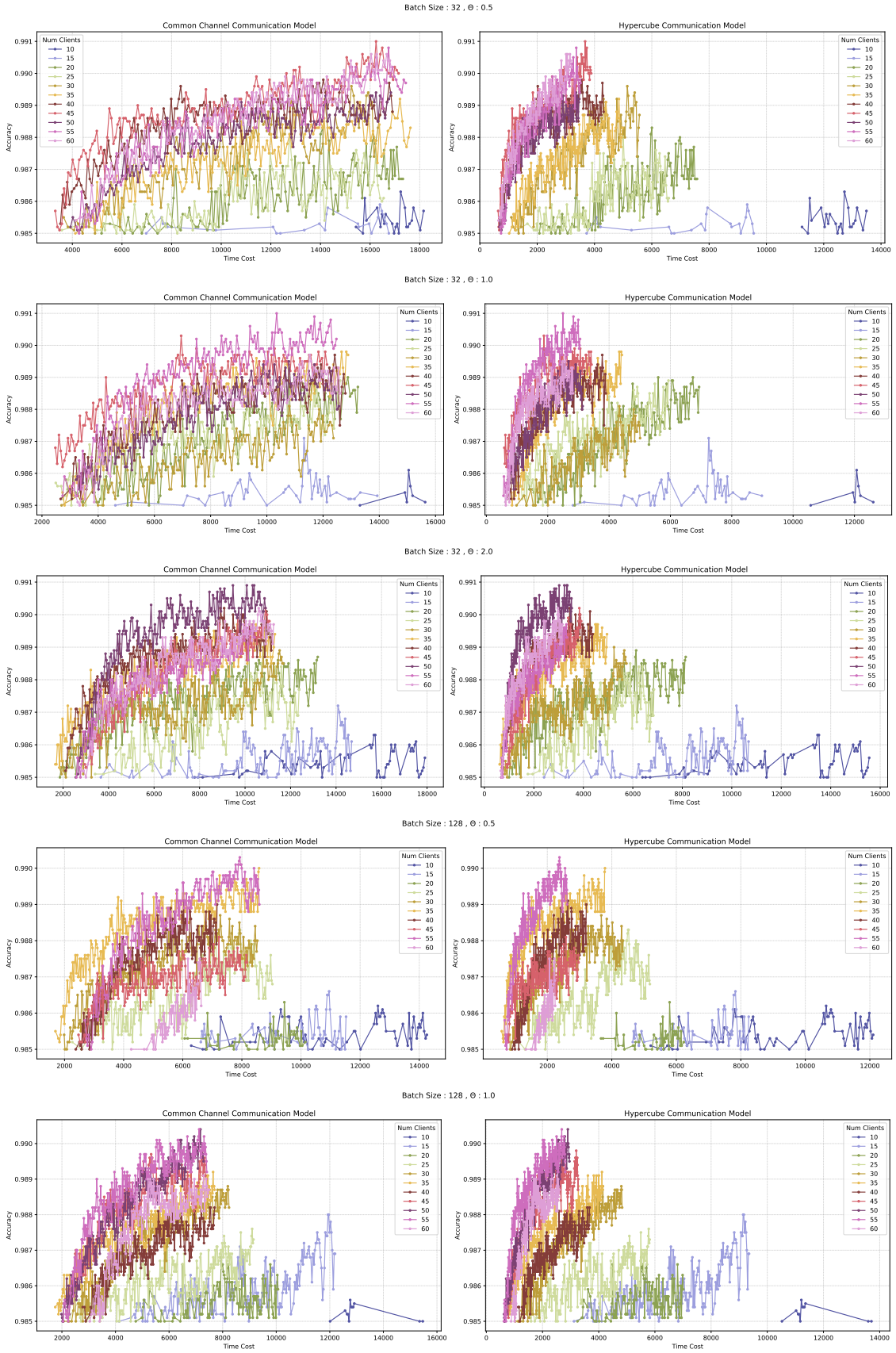



Figure A.7: Naive FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.985. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

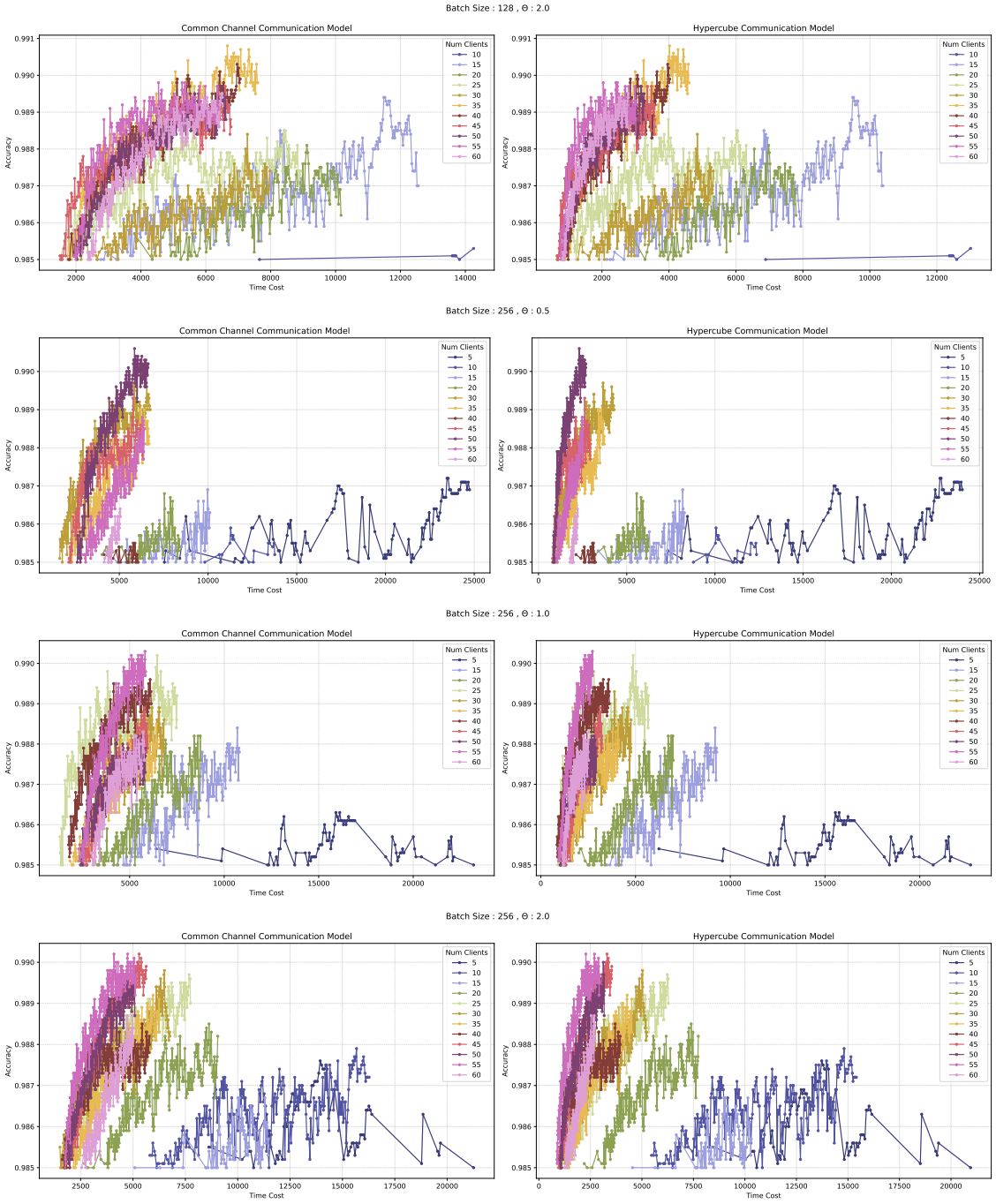



Figure A.7: Naive FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.985. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

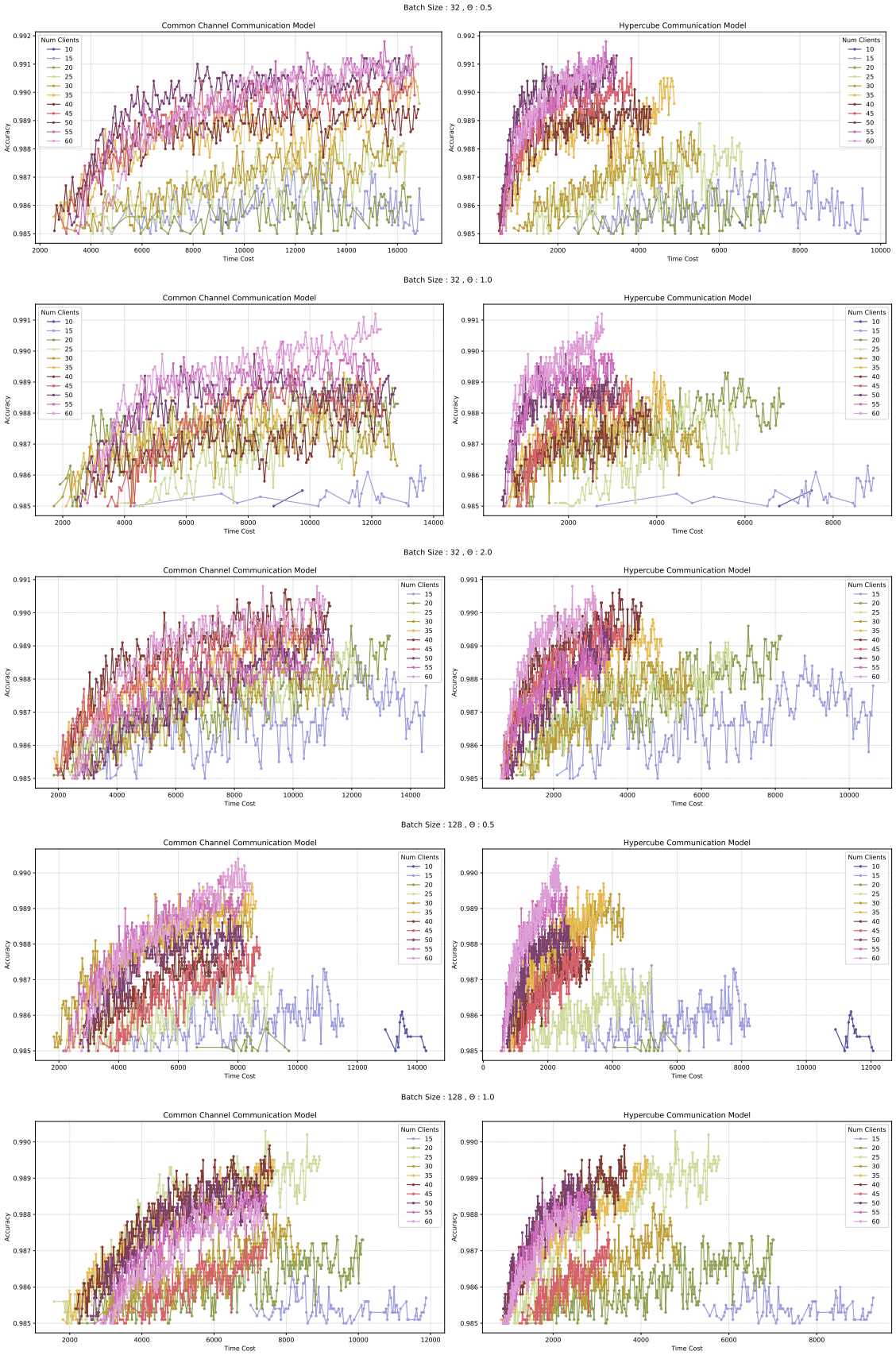



Figure A.8: Linear FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.985. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

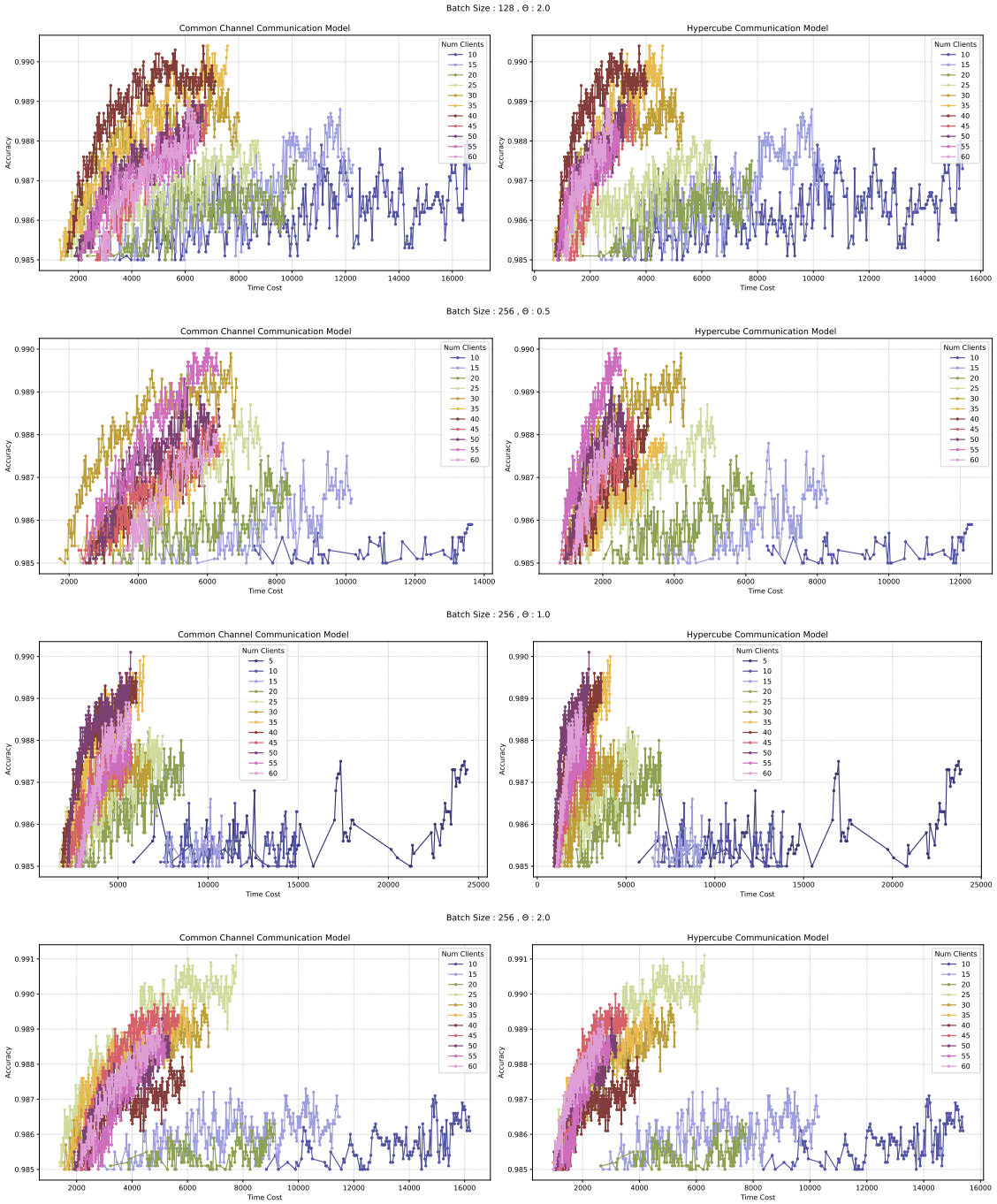



Figure A.8: Linear FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.985. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

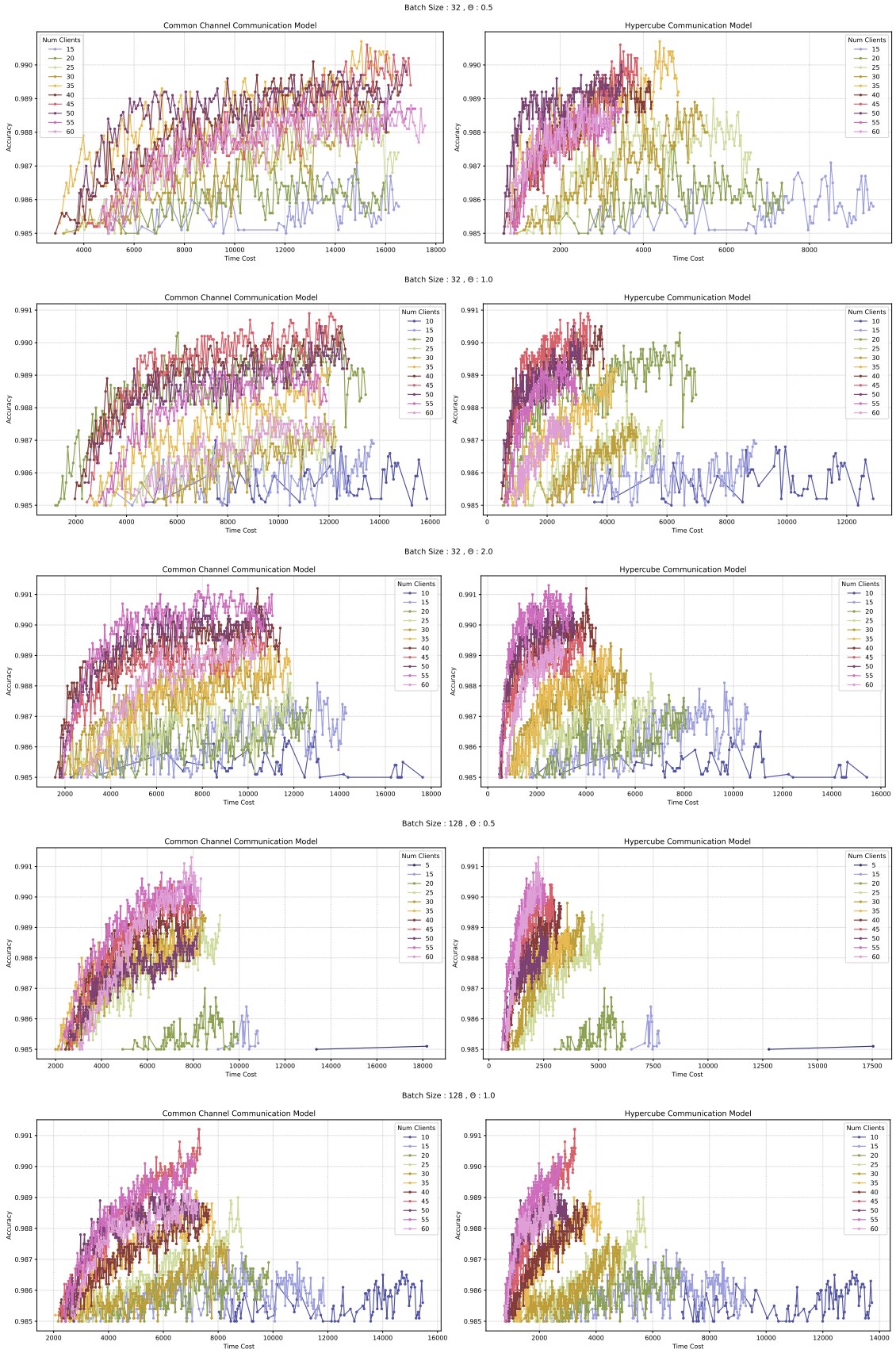



Figure A.9: Sketch FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.985. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

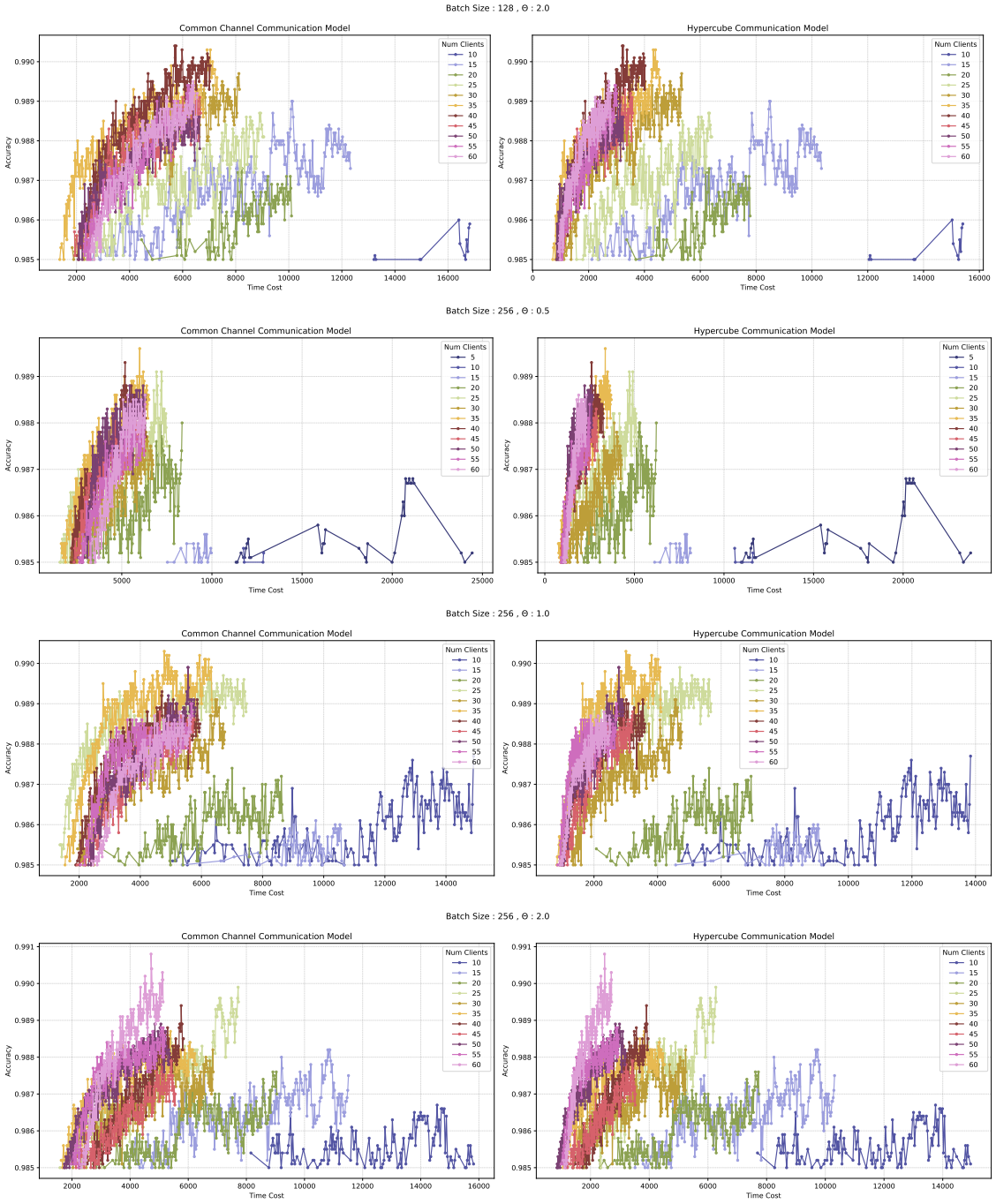



Figure A.10: Sketch FDA. Each chain of line segments represents a simulation's progression in time for a specific number of clients ($K$), after reaching the accuracy target of 0.985. Each dot within a chain represents a specific epoch within the simulation, capturing the time cost and accuracy at that moment.

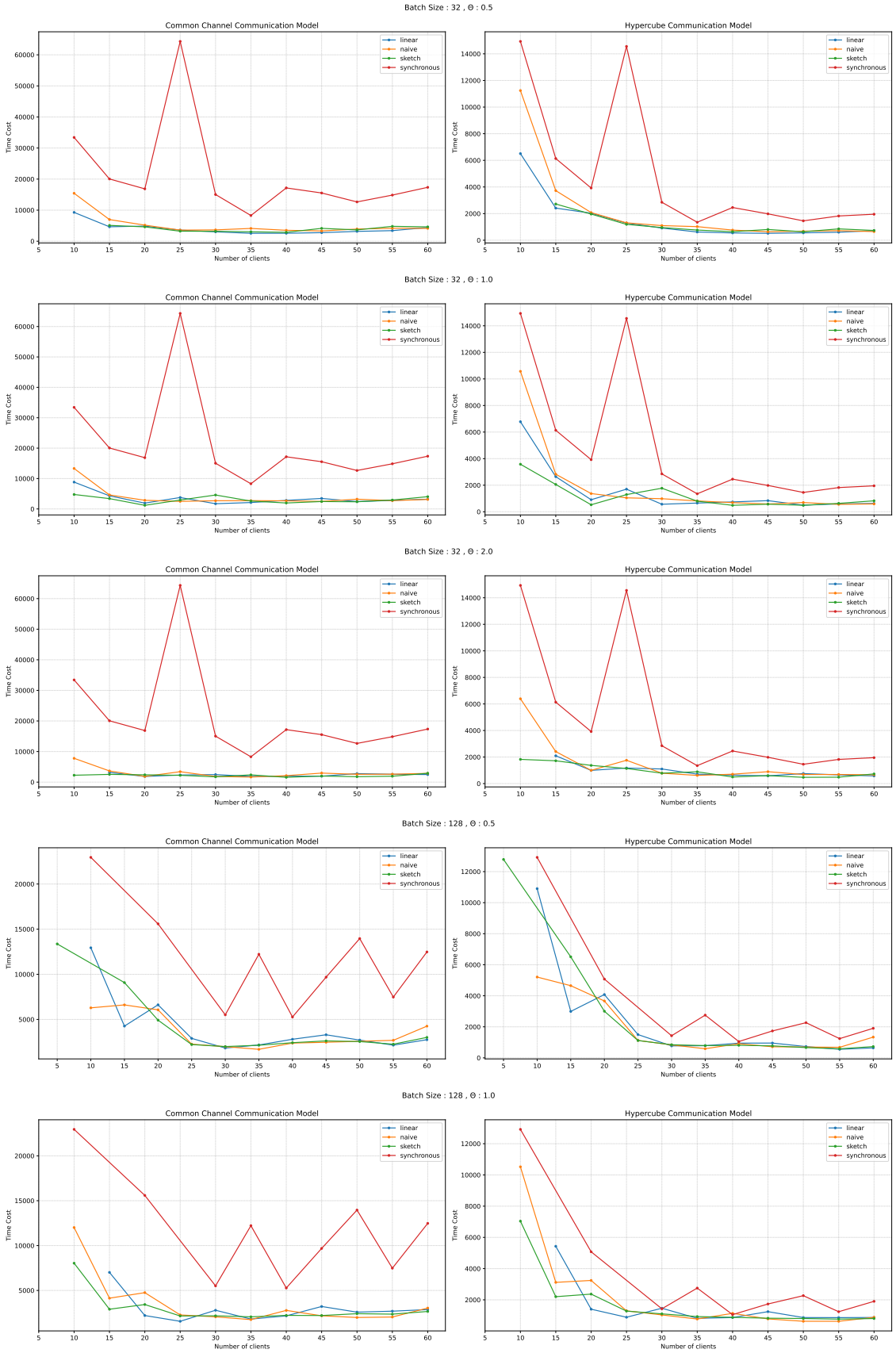



Figure A.11: *Filtered* (Section 4.1.5). Each dot within a chain of line segments represents an individual simulation, capturing the time cost incurred after reaching the accuracy threshold of 0.985. We illustrate the trends of the various methods in respect to the number of clients ($K$).

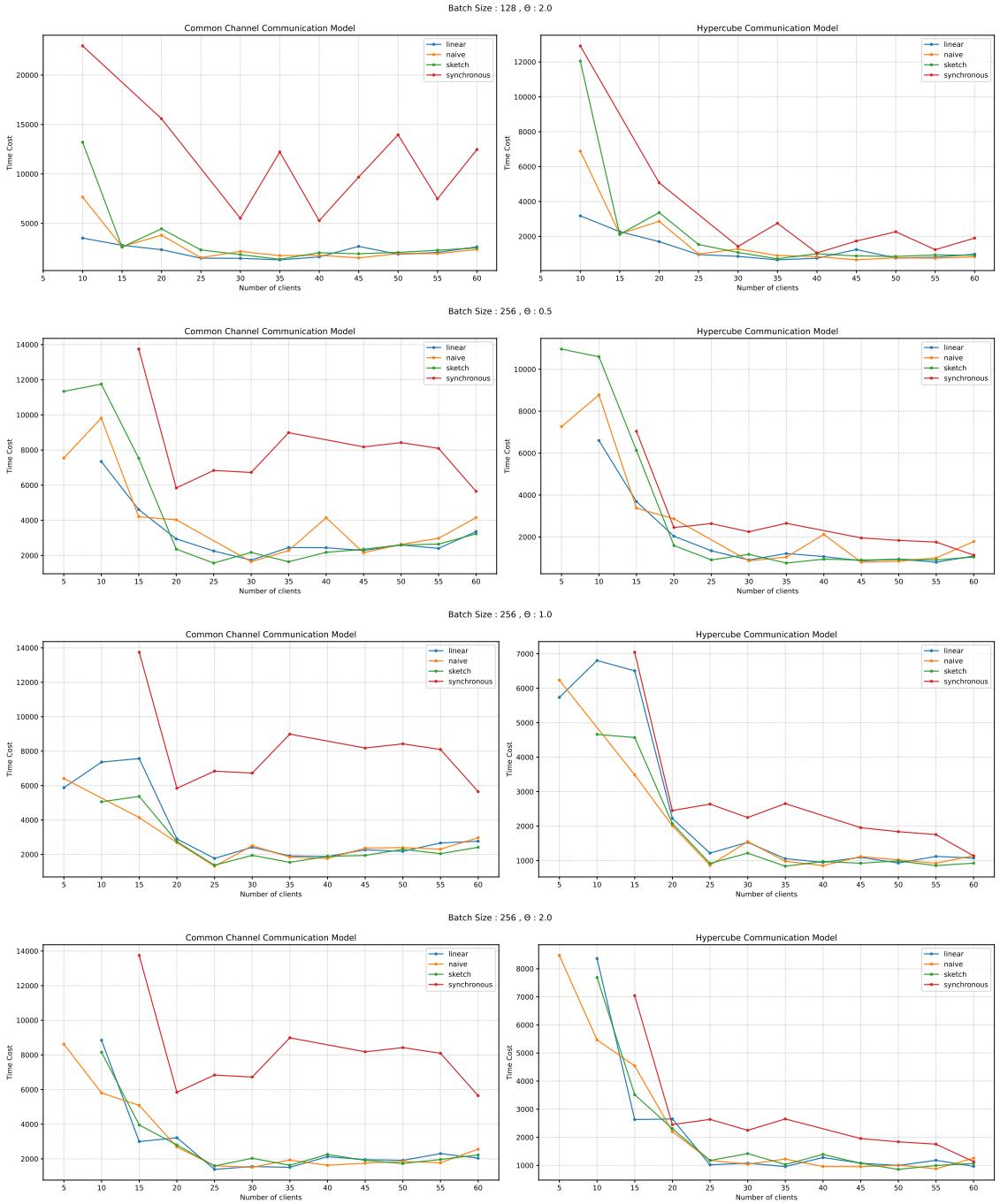



Figure A.11: *Filtered* (Section 4.1.5). Each dot within a chain of line segments represents an individual simulation, capturing the time cost incurred after reaching the accuracy threshold of 0.985. We illustrate the trends of the various methods in respect to the number of clients ($K$).

# Bibliography


[1] 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.

[2] Micah J. Sheller et al. "Federated learning in medicine: facilitating multi-institutional collaborations without sharing patient data". In: *Scientific Reports* 10.1 (July 2020), p. 12598. ISSN: 2045-2322. DOI: 10.1038/s41598-020-69250-1. URL: https://doi.org/10.1038/s41598-020-69250-1.

[3] H. Brendan McMahan et al. "Federated Learning of Deep Networks using Model Averaging". In: *CoRR* abs/1602.05629 (2016). arXiv: 1602.05629. URL: http://arxiv.org/abs/1602.05629.

[4] Peter Kairouz et al. "Advances and Open Problems in Federated Learning". In: *CoRR* abs/1912.04977 (2019). arXiv: 1912.04977. URL: http://arxiv.org/abs/1912.04977.

[5] Izchak Sharfman, Assaf Schuster, and Daniel Keren. "A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 301–312. ISBN: 1595934340. DOI: 10.1145/1142473.1142508. URL: https://doi.org/10.1145/1142473.1142508.

[6] Izchak Sharfman, Assaf Schuster, and Daniel Keren. "Aggregate Threshold Queries in Sensor Networks". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pp. 1–10. DOI: 10.1109/IPDPS.2007.370297.

[7] Vasilis Samoladas and Minos Garofalakis. "Functional Geometric Monitoring for Distributed Streams". In: *22nd International Conference on Extending Database Technology*. 2019, pp. 85–96. DOI: 10.5441/002/edbt.2019.09. URL: https://doi.org/10.5441/002/edbt.2019.09.

[8] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. "Sketch-based geometric monitoring of distributed stream queries". In: *Proceedings of the VLDB Endowment* 6 (Aug. 2013), pp. 937–948. DOI: 10.14778/2536206.2536220.

[9] Minos Garofalakis and Vasilis Samoladas. "Distributed Query Monitoring through Convex Analysis: Towards Composable Safe Zones". In: *20th International Conference on Database Theory (ICDT 2017)*. Ed. by Michael Benedikt and Giorgio Orsi. Vol. 68. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 14:1–14:18. ISBN: 978-3-95977-024-8. DOI: 10.4230/LIPIcs.ICDT.2017.14. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7066.

[10] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[11] Herbert Robbins and Sutton Monro. "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586.

[12] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[13] Nikhil Buduma and Nicholas Locascio. *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491925612.

[14] Jing An, Jianfeng Lu, and Lexing Ying. "Stochastic modified equations for the asynchronous stochastic gradient descent". In: *Information and Inference: A Journal of the IMA* 9.4 (Nov. 2019), pp. 851–873. DOI: 10.1093/imaiai/iaz030. URL: https://doi.org/10.1093%2Fimaiai%2Fiaz030.

[15] Yuhua Zhu and Lexing Ying. *A Sharp Convergence Rate for the Asynchronous Stochastic Gradient Descent*. 2020. arXiv: 2001.09126 [math.NA].

[16] Wei Zhang et al. *Staleness-aware Async-SGD for Distributed Deep Learning*. 2016. arXiv: 1511.05950 [cs.LG].

[17] Martin Zinkevich et al. "Parallelized Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper_files/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf.

[18] Wei Zhang et al. *A Highly Efficient Distributed Deep Learning System For Automatic Speech Recognition*. 2019. arXiv: 1907.05701 [eess.AS].

[19] Zachary Charles and Dimitris Papailiopoulos. *Gradient Coding via the Stochastic Block Model*. 2018. arXiv: 1805.10378 [stat.ML].

[20] Swanand Kadhe, O. Ozan Koyluoglu, and Kannan Ramchandran. *Gradient Coding Based on Block Designs for Mitigating Adversarial Stragglers*. 2019. arXiv: 1904.13373 [cs.IT].

[21] Luis Maßny et al. *Nested Gradient Codes for Straggler Mitigation in Distributed Machine Learning*. 2022. arXiv: 2212.08580 [cs.IT].

[22] Songze Li et al. *Near-Optimal Straggler Mitigation for Distributed Gradient Methods*. 2017. arXiv: 1710.09990 [cs.IT].

[23] Chen Chen et al. "Semi-dynamic load balancing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, Oct. 2020. DOI: 10.1145/3419111.3421299. URL: https://doi.org/10.1145%2F3419111.3421299.

[24] Wei Zhang et al. *Loss Landscape Dependent Self-Adjusting Learning Rates in Decentralized Stochastic Gradient Descent*. 2021. arXiv: 2112.01433 [cs.LG].

[25] Mehryar Mohri, Gary Sivek, and Ananda Theertha Suresh. *Agnostic Federated Learning*. 2019. arXiv: 1902.00146 [cs.LG].

[26] Reza Shokri and Vitaly Shmatikov. "Privacy-Preserving Deep Learning". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1310–1321. ISBN: 9781450338325. DOI: 10.1145/2810103.2813687. URL: https://doi.org/10.1145/2810103.2813687.

[27] Peter Sanders et al. *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. 1st. Springer Publishing Company, Incorporated, 2019. ISBN: 3030252086.

[28] George Karystinos. *Statistical Modeling and Pattern Recognition*. Lecture notes. Lectures 19-20, delivered on 2023-04-29 and 2023-05-30. 2023.

[29] F. Rosenblatt. *The perceptron - A perceiving and recognizing automaton*. Tech. rep. 85-460-1. Ithaca, New York: Cornell Aeronautical Laboratory, Jan. 1957.

[30] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[31] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural 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/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[33] C. Van Der Malsburg. "Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms". In: *Brain Theory*. Ed. by Günther Palm and Ad Aertsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 245–248. ISBN: 978-3-642-70911-1.

[34] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[35] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 9781605589077.

[36] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13-15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[37] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[38] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN: 9780262035613. URL: https://books.google.co.in/books?id=Np9SDQAAQBAJ.

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

[40] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC].

[41] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[42] François Chollet et al. *Keras*. https://keras.io. 2015.

[43] Samoladas and Kornidaris. "Extreme-Scale Online Machine Learning On Stream Processing Platforms". Unpublished Manuscript. 2023.

[44] 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.

[45] 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.

[46] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: `https://dask.org`.

[47] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[48] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: `https://proceedings.mlr.press/v9/glorot10a.html`.

[49] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: `http://yann.lecun.com/exdb/mnist/`.