CEA Comité Español de Automática
XLV JORNADAS AUTOMÁTICA MÁLAGA 2024
UNIVERSIDADE DA CORUÑA

Jornadas de Automática

A distributed learning proposal to improve industrial processes maintaining data privacy

Melgarejo Aragón, Marco.[a,*]

[a] *Centro Tecnológico de Componentes-CTC, Scientific and Technological Park of Cantabria (PCTCAN), 39011 Santander, Spain.*

To cite this article: Melgarejo Aragón, Marco. 2024. A distributed learning proposal to improve industrial pro-cesses maintaining data privacy. Jornadas de Automática, 45. https://doi.org/10.17979/ja-cea.2024.45.10976

Abstract

A distributed learning algorithm has been developed, focused on leveraging valuable information from industrial processes of various clients. This algorithm significantly improves the predictive capabilities of Machine Learning models by allowing access to a larger pool of training data. This is achieved by sharing the weights of the models among different participants, without the need to exchange the data itself, ensuring that each client maintains the privacy and security of their information. Thus, this approach not only optimizes the performance of the models individually but also enhances the overall level of artificial intelligence applied in the industrial sector.

Keywords: Machine Learning, Distributed Optimisation for Large-Scale Systems, Secure Networked Control Systems, Control under Communication Constraints.

1. Introduction

1.1. Current overview

As digitization and globalization progress, the volume of available data continues to grow. Concurrently, the complexity of artificial intelligence models designed to derive new insights from this data is also expanding. This complexity stems from the intricate challenges these models address in various domains, including computer vision, natural language processing, and speech recognition.

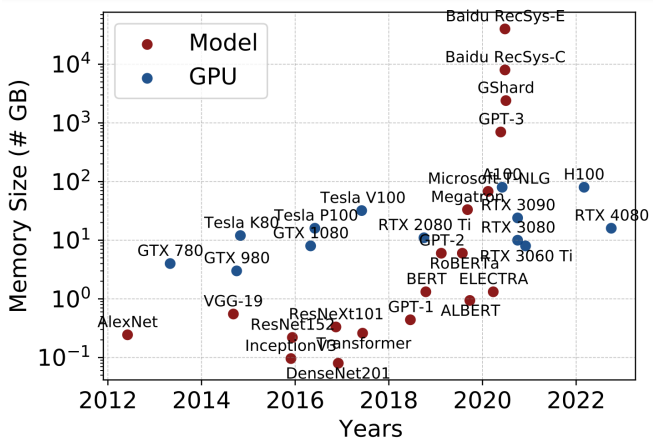
Training such complex models necessitates extensive training datasets and parameters to effectively enhance their inferencing and predictive capabilities. Thus, these two aspects continually feed into each other in an endless loop. The main limitation lies in the processing capacity for handling these complex models and the vast amounts of data they require (Tang et al., 2020).

Training these models involves different types of processing units, which can be categorized into Central Processing Units (CPUs), Graphics Processing Units (GPUs), and Tensor Processing Units (TPUs). CPUs are general-purpose processors that offer versatility for a broad range of computing tasks. GPUs excel in graphic-intensive and parallel processing tasks, handling vast amounts of data efficiently. TPUs are specially designed to manage tensor operations, which are data structures essential for tasks in Machine Learning (ML).

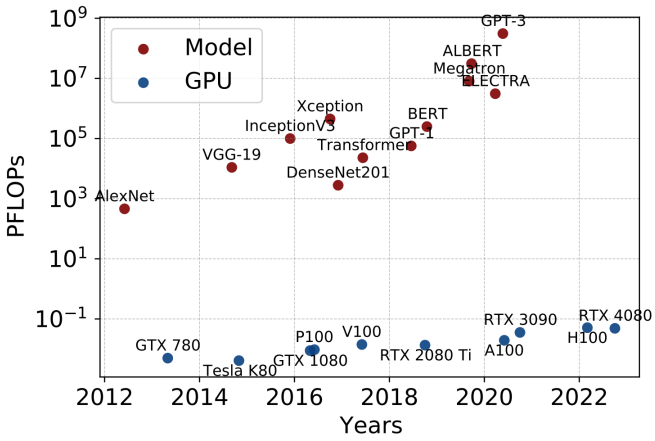
The training process increasingly involves significant computational costs and substantial time consumption. For instance, training a state-of-the-art ResNet-50 model on the well-known ImageNet database using a cutting-edge GPU like the Nvidia Tesla V100 takes roughly two days. A GPU's processing capacity is determined by its memory, used to store the necessary data for parallel computations, and its computational speed, measured in floating-point operations per second (FLOPS) (Tang et al., 2020).

To better understand the processing capacity challenges in the data science realm, consider two graphs in Figure 1. Figure 1a highlights the recent growth in GPU memory size alongside the memory requirements of leading-edge deep learning (DL) models, while Figure 1b displays petaFLOPS on the vertical axis and years on the horizontal to chart the evolution of both top-tier DL models and GPUs. The memory needed by models began to significantly increase in 2018 and surpassed the capabilities of GPUs by 2020 with the notable DL model GPT-3. In terms of FLOPS, the computational speed required by these DL models is immense compared to GPUs.

*Correspondence: mmelgarejo@centrotecnologicoctc.com



GPU memory and neural networks.



GPU and training model FLOPS.

Figure 1: Observed trends in GPU development and the progression of neural networks. Adapted from (Tang et al., 2020).

It's clear that the development of GPUs in terms of computational speed and memory is lagging and insufficient to handle the increasingly complex neural network models being developed, as shown in Figure 1.

To address this issue, two popular solutions exist:

- Developing highly optimized software, such as the cuDNN library, which enhances GPU performance and memory usage by providing efficient implementations of computational routines, utilizing specific GPU resources, and minimizing memory overhead.

- Employing distributed learning to accelerate the training process using multiple processors, including CPUs, GPUs, and TPUs. With distributed learning, spreading the workload can reduce overall training time. However, a new challenge arises: communication costs between processors hinder efficient scalability. In a collaborative network, the larger the network, the more data need to be transmitted among them (input data, model parameters, model updates, results of intermediate calculations, etc.), thus reducing the available bandwidth for communication.

Distributed learning refers to the method of training ML models by distributing the workload across multiple computing units or clients. There are several classical approaches to implementing distributed learning, including *data parallelism* and *model parallelism* (Mayer and Jacobsen, 2020):

- **Data parallelism:** This approach involves running the same model on different subsets of the data across multiple processors. Each computing worker receives a copy of the model parameters and computes local gradients or model updates using different mini-batches of data during each iteration. The local results are then shared between workers, aggregated, and broadcasted to update the global model. Data parallelism is a popular technique due to its significant scalability benefits.

- **Model parallelism:** Unlike data parallelism, model parallelism splits the model across various processors, with each part of the model being trained on the entire dataset. This method involves partitioning the model parameters among multiple computing workers, where each worker manages different parameters or layers of the model. Although model parallelism can be beneficial in certain cases, this work primarily emphasizes data parallelism techniques due to their extensive use and scalability advantages.

However, in the industrial sector, where the protection of proprietary information is paramount, distributed learning can play a crucial role. This will be detailed in the next section.

1.2. Specific problem

This work introduces an advanced distributed learning algorithm designed to optimize the use of valuable industrial process data from various clients. By leveraging Federated Learning (FL), the algorithm enhances the predictive power of ML models without requiring the direct sharing of sensitive or proprietary data. In this way, the centralization of data is avoided, which would have been the standard approach until now without this new approach. Even with the use of encryption or new technologies like blockchain, centralizing data requires the data to travel outside the client's domain.

Instead, it utilizes a novel approach where model weights are shared among the participants, allowing all clients to benefit from improved model accuracy while maintaining strict data privacy. This method ensures that each participant's data remains within their control, reducing risks associated with data breaches and ensuring compliance with industry-specific regulations.

Although we focus on the industrial domain, this work provides a step-by-step Python implementation of FL that is applicable to any field and type of dataset.

Such a strategy not only elevates the individual performance of ML models but also boosts the collective application of artificial intelligence within the industrial sector. By focusing on the aggregation of model updates rather than raw data exchange, FL offers a scalable solution that addresses both privacy concerns and the need for robust data utilization in complex industrial environments.

Therefore, while it is less likely, industrial data might contain personal information that must be handled according to strict privacy regulations, such as the GDPR (General Data Protection Regulation). This distributed learning approach ensures compliance with such regulations by keeping sensitive data within the client's domain and only sharing the necessary model updates (Su et al., 2022).

2. Methodology and development

2.1. Data source

The distributed learning process has been simulated with two canonical datasets, EMNIST and CIFAR-10, downloaded from TensorFlow/Datasets, a library of public datasets ready to use within the TensorFlow ecosystem in Python. The datasets are described below:

EMNIST Dataset

The EMNIST (Extended MNIST) dataset (TensorFlow, 2023b) is an extension of the well-known MNIST (Modified National Institute of Standards and Technology) dataset, designed to include letters in addition to the digits from 0 to 9. EMNIST contains grayscale images of handwritten letters, both uppercase and lowercase, as well as digits. The dataset size consists of 697932 training images and 116323 testing images. Like MNIST, each image in EMNIST is 28x28 pixels. The images are in grayscale, with pixel values ranging from 0 (white) to 255 (black). Figure 2 shows some sample images from the EMNIST dataset.

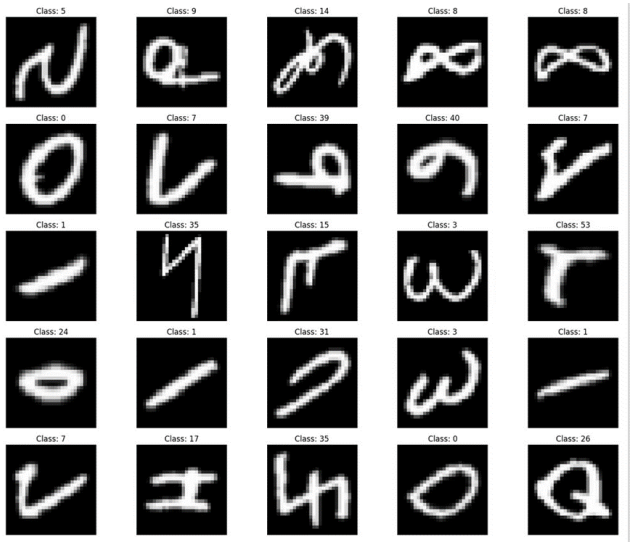


Figure 2: Sample images from the EMNIST dataset.

CIFAR-10 Dataset

The CIFAR-10 dataset (TensorFlow, 2023a) is widely used in the field of ML and computer vision to evaluate image classification algorithms. It consists of 60000 color images of 32x32 pixels, divided into 10 classes, with 6000 images per class. The classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Figure 3 shows examples of these images. The dataset is divided into 50000 training images and 10000 test images.

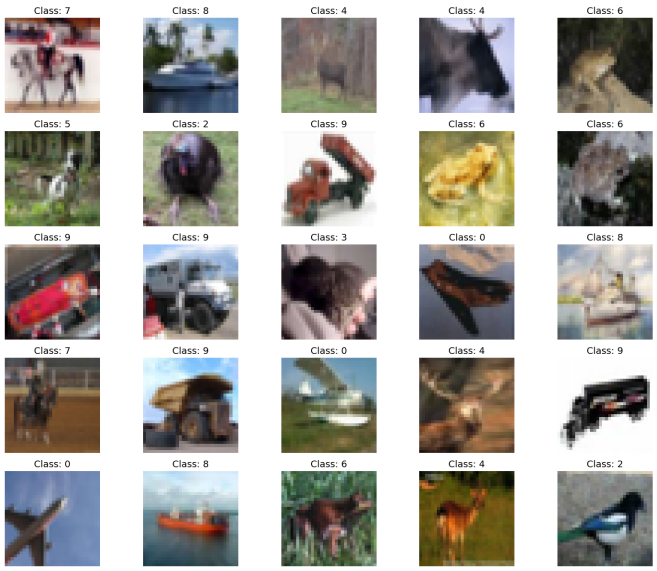


Figure 3: Sample images from the CIFAR-10 dataset.

2.2. Distributed architecture

A distributed learning architecture with a central server has been simulated using synthetic clients. The scheme followed by this architecture is shown in Figure 4, where an example with 5 synthetic clients is depicted. Additionally, its pseudocode is shown in Code Block 1, which uses the aggregation function, with pseudocode shown in Code Block 2.

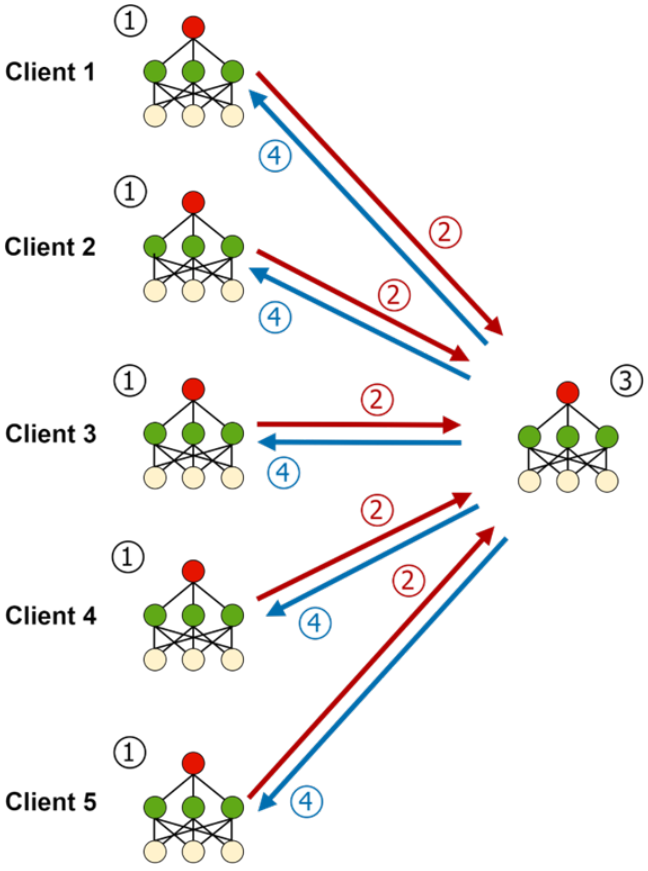


Figure 4: Example of a Federated Learning architecture with a central server and five synthetic clients.

```python
def FL_architecture(n_times, initial_weights, n_i, model, data_dict):
    """
    Federated learning architecture implementation.

    :param n_times: Total number of communication rounds
    :param initial_weights: Initial model weights from the trained ANN
    :param n_i: List indicating the number of samples per client
    :param model: Pre-trained ANN model
    :param data_dict: Dictionary containing clients' data with training and testing splits
    :return save_weights: Aggregated model weights from all rounds
    """
    # Initialize list to store weights from all rounds
    save_weights = []
    # Iterate over communication rounds
    for i in range(1,n_times+1):

        # Initialize list to collect client weights for this round
        weights_round = list()

        # Generate weights for each client
        for name in data_dict.keys():
            # Assign initial weights to the client model
            model.set_weights(initial_weights)
            # Train the client's model
            X_train_np = np.array(data_dict[name]['X_train'])
            y_train_np = np.array(data_dict[name]['y_train'])
            # Train the model with the given parameters
            model.fit(X_train_np, y_train_np, epochs=2, batch_size=128)
            # Retrieve the trained weights from the client model
            weights_client = model.get_weights()
            # Store the client's weights
            weights_round.append(weights_client)

        # Perform aggregation of the client weights
        avg_weights = ave_weights(n_i, weights_round)
        # Update the global model with aggregated weights
        model.set_weights(avg_weights)
        # Update initial weights for the next communication round
        initial_weights = avg_weights

        # Store the aggregated weights of this round
        save_weights.append(avg_weights.copy())
    return save_weights
```

Code Block 1: Python code for a simulated Federated Learning architecture algorithm.

```python
def ave_weights(n_i, listOfWeights):
    """
    Aggregation function to compute weighted average of model parameters.

   :param list n_i: Number of samples per client
   :param list listOfWeights: Weights from each client
   :return: Weighted average of the model parameters for the global model
    """
    N = sum(n_i) # Total number of samples across all clients
    # Initialize the global model weights to zero
    ave_weights = listOfWeights[0]
    ave_weights = [i * 0 for i in ave_weights]
    # Iterate over each client
    for j in range(len(n_i)):
        # Retrieve the weights from the current client
        rec_weight = listOfWeights[j]
        # Scale the weights by the number of samples in the client's dataset
        rec_weight =  [i * n_i[j] for i in rec_weight]
        # Normalize by the total number of samples
        rec_weight =  [i / N for i in rec_weight]
        # Add the scaled weights to the cumulative total
        ave_weights = [x + y for x, y in zip(ave_weights, rec_weight)]
    return ave_weights
```

Code Block 2: Python code for an aggregation function to compute a weighted average of client model parameters in Federated Learning.

This Figure 4 is labelled with numbers corresponding to the following steps:

1. Training a ML model on each client.
2. Extracting model weights from clients.
3. Aggregating model weights into a global model.
4. Global model weights are sent back to each client.
5. Repeating the last steps for more communication rounds.

The Code Block 1 demonstrates a simulated implementation of a FL architecture in Python. This code block illustrates how the FL process can be simulated using an artificial neural network (ANN) and a central server that coordinates the training of multiple synthetic clients. The `FL_architecture` function is responsible for managing this process over several rounds of communication.

The `FL_architecture` function starts by initializing a list, `save_weights`, which will store the aggregated model weights at the end of each communication round. Then, it uses a "for loop" to iterate through the total number of communication rounds specified by the `n_times` parameter. Within each iteration of this loop, a list, `weights_round`, is initialized to collect the weights of the models trained by each client during that round.

For each client, identified by the keys in the `data_dict` dictionary, the code assigns the initial weights to the client's model using `model.set_weights(initial_weights)`.

Then, the client's training data is extracted from the `data_dict` dictionary and converted into NumPy arrays. The client's ANN model is trained using this data via the `model.fit` function, which fits the model to the training data for the given epochs with the provided batch size.

After training the client's model, the adjusted model weights are retrieved with `model.get_weights()` and stored in the `weights_round` list. This process is repeated for all clients in each communication round. Once all clients have trained their models and their weights have been collected, these weights are aggregated.

The aggregation of the client model weights is performed using the `ave_weights` function, detailed in Code Block 2. This function takes as input the list of client weights and the number of data samples from each client. The goal of this function is to calculate a weighted average of the weights, where each client's weight set is weighted according to the size of their dataset.

The `ave_weights` function begins by calculating the total number of data samples from all clients by summing the values of `n_i`. Then, it initializes the global model weights to zero. Next, it iterates over each client, retrieving the current client's weights and scaling them according to the number of samples in the client's dataset. These scaled weights are normalized by dividing by the total number of samples and are added cumulatively to the global weights. This process ensures that the final weights fairly reflect each client's contribution based on the size of their data.

Finally, the aggregated model weights for the current round are assigned to the global model with `model.set_weights(avg_weights)`, and the initial weights are updated for the next communication round. The aggregated weights of each round are copied and stored in the `save_weights` list, which is returned at the end of the `FL_architecture` function. These aggregated weights are ready to be used in the calculation of classification metrics on the test set, allowing the evaluation of the federated model's performance after each round of training and aggregation.

In summary, the presented code blocks show a detailed implementation of a FL algorithm, from data distribution among clients and local model training to weight aggregation and global model updating. This approach allows simulating and evaluating the efficiency and effectiveness of FL in a controlled environment using synthetic data.

2.3. Machine learning details

In order to obtain results focusing on the distributed learning architecture, a key aspect of this study, a basic academic ML architecture was chosen. This architecture is intentionally simple, with few layers and neurons per layer, to ensure that the complexity of the model does not overshadow the analysis. The models were trained with only 2 epochs and a batch size of 128. By using a straightforward ML architecture, the primary focus remains on evaluating the distributed learning process itself, making the improvements in model performance more evident.

The Artificial Neural Networks (ANN) that were run in both use cases had the architecture shown below:

- **Conv2D layer.** Filters 32. Kernel size: (3,3). Activation: *ReLU*. Input shape: (28, 28, 1) (EMNIST dataset) or (32, 32, 3) (CIFAR-10 dataset).
- **MaxPooling2D layer.** Pool size: (2,2). Strides: None.
- **Conv2D layer.** Filters 64. Kernel size: (3,3). Activation: *ReLU*.
- **Flatten layer.**
- **Dense layer.** Units 64. Activation: *ReLU*.
- **Dense layer.** Units 62 (EMNIST dataset) or units 10 (CIFAR-10 dataset). Activation: *SoftMax*.

The employed loss function was the *categorical cross-entropy loss*, mathematically defined in the Equation 1 and the model's performance was evaluated based on the accuracy metric.

$$\mathcal{L}(y, p) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij} \cdot \log(p_{ij}) \tag{1}$$

where y_{ij} indicates whether sample i belongs to class j (1 if true, 0 otherwise), p_{ij} is the predicted probability by the model for sample i belonging to class j, N is the total number of samples, and K is the number of classes.

For both use cases, the original train/test split from TensorFlow, previously mentioned, was used. The pixel values were normalized by dividing by 255.

It is important to emphasize that no validation split is needed to run the models during the distributed communication. This is because, in a real-world scenario, it is proposed that before the communication rounds, the client with the most balanced and extensive dataset, characterized by a high degree of IID (Independent and Identically Distributed) data (Zhang et al., 2021), designs the optimal ANN architecture. Therefore, only testing measures will need to be run by each client on each round.

2.4. Software

The main dependencies are shown below:

- Python version: 3.8.10
- NumPy version: 1.23.4
- Pandas version: 2.0.3
- Matplotlib version: 1.23.4
- Scikit-learn version: 1.3.2
- TensorFlow version: 2.11.0

3. Results and discussion

For reproducibility of results, the following seeds were fixed to zero value: `numpy.random.seed`, `random.seed` and `tf.random.set_seed`. The EMNIST dataset was used to simulate a FL architecture of 5 communication rounds with 3 sintetic clients and the same for CIFAR-10 dataset but duplicating the number of rounds in order to reach a metric convergence when testing, their results are shown in the Figure 5 and Figure 6 respectively.

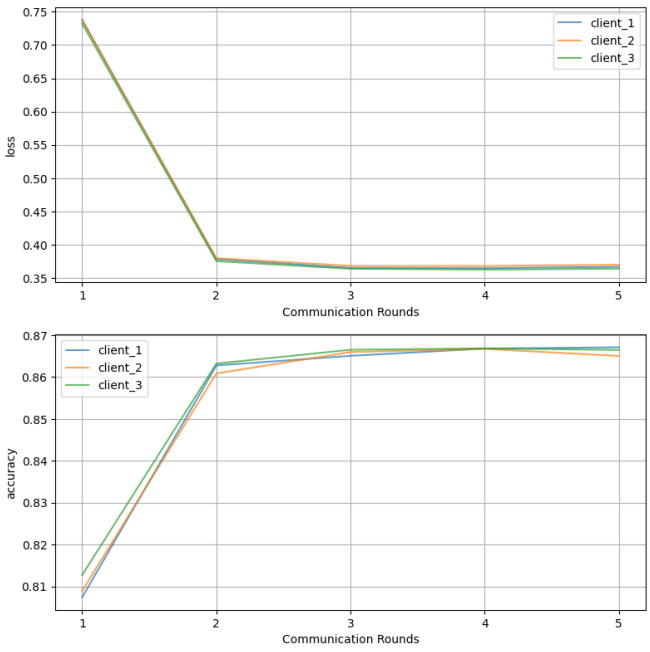


Figure 5: Evolution of the loss and accuracy metrics for the EMNIST dataset.

In Figure 5, it can be seen that in this simulation, the loss and accuracy metrics evolve with very similar values. After only 3 communication rounds, they converge to stable values, significantly improving the loss.

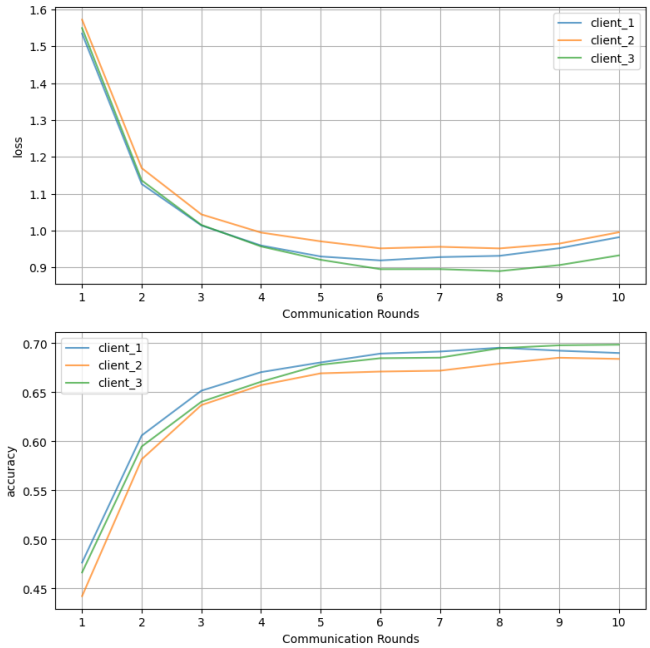


Figure 6: Evolution of the loss and accuracy metrics for the CIFAR-10 dataset.

Additionally, in Figure 6, it can be seen that the metrics evolve rapidly and favorably, converging for all synthetic clients in about 8 communication rounds.

The simulation of distributed learning communication conducted in this study, with the two use case datasets, shows promising results and serves as a demonstration that this could be exploited in the real-world scenario. The implementation of this approach would enable harnessing the vast amount of data generated by daily operations without compromising the privacy and security of information. Furthermore, by distributing data processing across multiple nodes or devices, companies could perform complex analyses and train artificial intelligence models efficiently and scalably.

This demonstration opens the door to collaborative projects that could not only improve individual industrial processes but also contribute to the overall well-being of the sector through the development of more accurate and efficient models.

Although there are comprehensive tools in the state of the art, such as FATE (FederatedAI, 2024), that enable the development of distributed learning, their application has mainly focused on the finance sector. Nevertheless, there are still many challenges to address (Liu et al., 2024). A very significant limitation of such distributed learning tools could be the absence of a reliable mediator that facilitates security and privacy during information exchange. It is proposed here that the mediator be responsible for the central server on which the global model is formed. It is worth noting that for the successful implementation of a common ML model, standardization in data collection and processing among participating companies is required. This could be achieved through the adoption of regulations and procedures, such as those stipulated in certain ISO standards, to ensure the compatibility and relevance of the data used in the architecture. Furthermore, this would allow promoting a culture of standardization and quality in industrial production processes.

References

FederatedAI, 2024. FederatedAI (FATE): Federated Learning Framework for Secure and Privacy-Preserving AI. `https://federatedai.org/`.

Liu, Y., Kang, Y., Zou, T., Pu, Y., He, Y., Ye, X., Ouyang, Y., Zhang, Y.-Q., Yang, Q., 2024. Vertical federated learning: Concepts, advances, and challenges. IEEE Transactions on Knowledge and Data Engineering, 1–20.
URL: `http://dx.doi.org/10.1109/TKDE.2024.3352628`
DOI: `10.1109/tkde.2024.3352628`

Mayer, R., Jacobsen, H.-A., 2020. Distributed and parallel machine learning: A comprehensive review. Journal of Parallel and Distributed Computing.

Su, W., Li, L., Liu, F., He, M., Liang, X., 2022. Ai on the edge: a comprehensive review. Artificial Intelligence Review 55 (8), 6125–6183.

Tang, Z., Shi, S., Chu, X., Wang, W., Li, B., 2020. Communication-efficient distributed deep learning: A comprehensive survey. CoRR abs/2003.06307.
URL: `https://arxiv.org/abs/2003.06307`

TensorFlow, 2023a. Cifar-10 dataset in tensorflow datasets.
URL: `https://www.tensorflow.org/datasets/catalog/cifar10`

TensorFlow, 2023b. Emnist dataset in tensorflow datasets.
URL: `https://www.tensorflow.org/datasets/catalog/emnist`

Zhang, W., Wang, X., Zhou, P., Wu, W., Zhang, X., 2021. Client selection for federated learning with non-iid data in mobile edge computing. IEEE Access 9, 24462–24474.
DOI: `10.1109/ACCESS.2021.3056919`