Facultad de ciencias

COMPARACIÓN DE TÉCNICAS DISTRIBUIDAS DE APRENDIZAJE AUTOMÁTICO APLICADAS A DATOS MÉDICOS DISPONIBLES EN ABIERTO

(Comparison of distributed machine learning techniques applied to openly available medical data)

Trabajo de Fin de Máster
para acceder al

Máster en Ciencia de Datos

Autor: Marco Antonio Melgarejo Aragón

Directora: Judith Sáinz-Pardo Díaz

Co-director: Álvaro López García

Febrero 2024

Contents

Abstract

Distributed machine/deep learning refers to algorithms and systems designed to enhance performance, preserve privacy, and scale to larger training data and models. The aim of this study is to compare the performance of different distributed machine learning techniques, such as federated learning, gossip learning, or ring all-reduce architecture. To achieve this, their application is proposed using artificial neural networks on an openly available medical dataset. Various metrics will be evaluated based on the architecture configuration and the number of rounds carried out. The implementation of the three architectures using Python is proposed in a scenario where data distribution is simulated. All implemented code can be openly accessed.

Keywords: Machine learning, distributed learning, federated learning, medical data, privacy.

Resumen

El aprendizaje automático/profundo distribuido se refiere a algoritmos y sistemas de aprendizaje automático/profundo diseñados para mejorar el rendimiento, preservar la privacidad y escalar a datos de entrenamiento y modelos más grandes. El objetivo de este trabajo es comparar el rendimiento de diferentes técnicas de aprendizaje automático distribuido, como el aprendizaje federado, el aprendizaje "por rumores" o la arquitectura de reducción total en anillo. Para ello, se propone su aplicación utilizando redes neuronales artificiales a un conjunto de datos médicos en abierto. Se evaluarán distintas métricas en función de la configuración de la arquitectura y el número de rondas llevadas a cabo. Se propone la implementación de las tres arquitecturas utilizando Python en un escenario donde se simula distribución de los datos. Todo el código implementado se puede consultar en abierto.

Palabras clave: Aprendizaje automático, aprendizaje distribuido, aprendizaje federado, datos médicos, privacidad.

Acknowledgements

I wish to express my deepest gratitude to my supervisor, Professor Judith Sáinz-Pardo Díaz, for her exceptional support, guidance, and patience, not only during the development of this Master's Thesis but also throughout my internship at IFCA.

Her dedication and effort in guiding me have been unparalleled, far exceeding what any other mentor has ever done for me. Therefore, I reiterate my sincere thanks.

Likewise, I extend my gratitude to the Centro Tecnológico CTC for granting me the opportunity to continue overcoming new challenges and for keeping my motivation alive, which is essential for tackling any project with enthusiasm. I deeply appreciate their interest in my training and development, especially for providing me with the necessary resources to complete this report.

Last but not least, I would like to thank my partner, Irene, for her unwavering companionship throughout these years, in both joyful moments and challenges. Words cannot fully express my gratitude and appreciation for your constant support.

Acronyms

AI Artificial Intelligence 23, 39, 50

ANN Artificial Neural Network 10, 12, 13, 17, 18, 20, 23, 27, 29–32, 34, 41, 44, 49

AUC Area Under the Curve 12, 16, 29, 31, 41, 42, 44

CPU Central Processing Unit 7, 8, 20, 39

CV Cross-Validation 16, 30, 31

DGC Deep Gradient Compression 20

DL Deep Learning 8, 10, 15, 23, 39

DLC Data Life Cycle 10, 11

EOSC European Open Science Cloud 39

FAIR Findable, Accessible, Interoperable, and Reusable 11

FL Federated Learning 9, 10, 13, 18–21, 31–35, 41–43

FLOPS Floating Point Operations per Second 7, 8

GAN Generative Adversarial Networks 20

GDPR General Data Protection Regulation 9, 10

GL Gossip Learning 9, 10, 13, 18, 19, 21, 35, 38, 41, 42, 44, 45

GPU Graphics Processing Unit 7, 8, 39

ML Machine Learning 9, 10, 13, 15, 23, 25, 27, 39

NaN Not a Number 25–27

RAR Ring All-Reduce 9, 10, 13, 19, 35, 37, 38, 41, 42, 44, 45

ROC Receiver Operating Characteristic 16

TPU Tensor Processing Unit 7, 8

Chapter 1

Introduction

1.1 General background

With digitalization and globalization, the amount of available data is constantly increasing. At the same time, the complexity of artificial intelligence models that extract new insights from this data is also increasing. This complexity arises from the intricate problems addressed in various fields such as computer vision, natural language processing, and speech recognition. Training these complex models require large training datasets and parameters to enhance their inference and predictive tasks effectively. Thus, these two fronts feed back into an endless loop. The limit resides in the processing capacity of these complex models and large amounts of data [1].

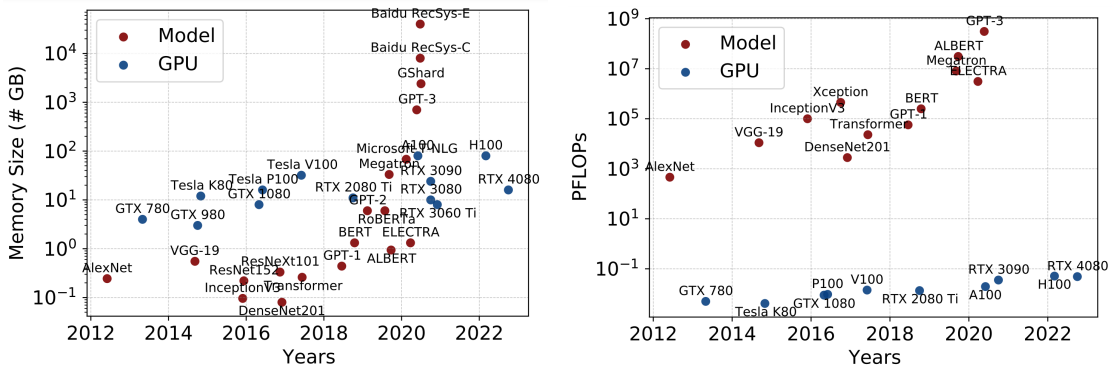
The training of these models requires processing units, which can be categorized into Central Processing Unit (CPU), Graphics Processing Unit (GPU), and Tensor Processing Unit (TPU). The CPU performs general-purpose computing calculations and can be very versatile. The GPU is more efficient for graphical tasks and parallel computing, capable of processing massive amounts of data. The TPU is a processor specifically designed to perform operations related to tensors, which are data structures used in machine learning and artificial intelligence.

Therefore, the training process increasingly involves significant computational costs and excessive time consumption. To illustrate, running a state-of-the-art ResNet-50 model for 90 epochs on the famous ImageNet database using a cutting-edge GPU like the Nvidia Tesla V100 takes approximately two days [1].

The processing capacity of a GPU is determined by its memory, which is used to store the data required for parallel computations, and its computing speed, measured in Floating Point Operations per Second (FLOPS). To further understand the context of the processing capacity issue in the world of data science, consider the two graphs in Figure 1.1.

The Figure 1.1a shows the increase over recent years in the memory size of GPUs and the required memory size by leading-edge Deep Learning (DL) models, while Figure 1.1b depicts the petaFLOPS on the ordinate axis and the years on the abscissa to illustrate the evolution of both top-tier DL models and GPUs. The memory size required by models began to grow significantly in 2018 and surpassed the limits of GPUs by 2020 with the famous DL model GPT-3. In terms of FLOPS, the quantity of computing speed needed by these DL models is enormous compared to GPUs.

It is evident that the development of GPUs in terms of computing speed and memory lags behind and it is insufficient to address the increasingly complex neural network models being developed, as shown in Figure 1.1.



(a) GPU memory and neural networks.

(b) GPU and training model FLOPS.

Figure 1.1: The patterns observed in GPU development and the evolution of neural networks. Adapted from the reference [1].

To overcome this problem, there are *two popular solutions: developing highly optimized software*, such as the cuDNN library, which optimizes GPU performance and memory usage by providing efficient implementations of computing routines, using specific GPU resources, and reducing memory overhead [2], or employing *distributed learning* to accelerate the training process using multiple processors, including CPUs, GPUs, and TPUs. With the latter, distributing the workload can reduce the total training time. However, a new challenge arises: the communication cost between processors hinders efficient scalability. In a collaborative network, the larger the network, the more data needs to be transmitted between them (input data, model parameters, model updates, intermediate calculation results, etc.), thus reducing the communication bandwidth available to each of them [1].

Distributed Learning is an approach where the training of machine learning models is distributed across multiple computing units or clients. This can be essential for:

- Handling large datasets, offering the scalability to accommodate more training data.
- Training complex models that are computationally intensive, thereby improving performance.
- Preserving the privacy of the data involved.

Classically, distributed learning can be implemented in various ways, such as *data parallelism* and *model parallelism* [3]:

- **Data parallelism:** This technique involves training the same model on different subsets of the data across multiple processors. The model parameters are replicated to all computing workers, and during each iteration, each worker computes the local gradient or model updates using different mini-batches of data. Results are then exchanged between workers, followed by aggregation and broadcast operations to obtain the new global model. Data parallelism is a widely employed distributed training technique and offers significant scalability benefits.

- **Model parallelism:** In contrast to data parallelism, model parallelism involves splitting the model across different processors, where each part of the model is trained on the entire dataset. This approach requires partitioning model parameters among multiple computing workers, with each worker holding different parameters or layers of the model. While model parallelism offers advantages in certain scenarios, this work primarily focuses on techniques associated with data parallelism due to its widespread usage and scalability benefits.

Moreover, traditional centralized machine learning approaches, typically employing a server with multiple processors, are used to manage the growing quantity of data mentioned before. However, these approaches face challenges related to data protection regulations such as the General Data Protection Regulation (GDPR), which impose strict rules on the handling of personal data. Additionally, there is a growing public awareness of privacy issues. Deploying centralized machine learning systems in data centers can also lead to high latency [4][5].

An important aspect of this work is that distributed learning techniques can address scenarios where privacy constraints prevent the centralization of data. One such approach, known as Federated Learning (FL), distributes model training across multiple devices or clients, each with its own local data. In this decentralized approach, data is not stored in a central server. By doing so, Federated Learning tackles the problem of privacy by ensuring that sensitive data remains on individual devices or servers that has generated or stored them, thereby reducing the risk of data breaches or privacy violations.

Other approaches to distributed learning with focus on data privacy, such as Gossip Learning (GL) or Ring All-Reduce (RAR), have been developed for similar purposes. These last approaches proposes architectures that share model parameters between the involved clients, without the need for a central server.

1.2 Specific problem

The present work addresses the topic of distributed learning in the context of privacy-preserving Machine Learning (ML), wherein data is distributed across different locations for privacy reasons or even technical reasons.

To board this, five distributed ML approaches have been addressed, including FL, RAR, GL. For implementing these approaches, the "Heart Disease" dataset has been chosen as the use case (detailed in Chapter 3). This dataset is significant due to its comprehensive nature, compiling a rich array of shared features from multiple datasets across various hospitals [6].

Its application in decentralized learning environments underscores the capacity of distributed machine learning methods to process sensitive medical data securely, meeting the rigorous privacy standards of regulations like GDPR. This suitability makes it an ideal choice for demonstrating the advanced capabilities and privacy adherence of these techniques.

1.3 Research questions and objectives

The study aims to compare different distributed ML/DL techniques, namely FL, RAR, GL, and a customized architecture. All of these are detailed later in Chapter 3. The research questions include evaluating the performance differences between these techniques and understanding their effectiveness in handling sensitive data like medical records.

The overall objective of this Master's thesis is to compare the performance of different distributed learning techniques. The specific objectives are:

- To investigate and develop distributed learning approaches.
- To apply the approaches to an openly available dataset with sensitive data and distinct clients, ensuring privacy in this simulated scenario.
- To prepare and analyze the dataset used.
- To search for an effective Artificial Neural Network (ANN) architecture for all the clients.
- To simulate the implementation the distributed learning architectures in Python, as well as related methods.

1.3.1 Methodology applied

The Data Life Cycle (DLC) offers a comprehensive view of the stages required for effectively managing and preserving data to ensure its utility and re-usability. One version of DLC recommended by Horizon Europe Program Guide [7] is the shown in Figure 1.2.

Figure 1.2: Data life cycle. Adapted from reference [8].

The traditional DLC entails the following stages [8][9]:

- *Planning*: Data management planning consists of defining the strategy that you plan to use for managing the data and the documentation generated within the research project.

- *Collection*: Data collection involves obtaining information about specific variables through various methods like questionnaires or patient records, focusing on data quality. It's also possible to reuse existing data, including previous datasets or curated resources.

- *Process*: The data processing phase converts and prepares data for analysis, automating format conversion, quality control, and preprocessing. Its goal is to obtain readable, high-quality data, discarding low-quality information. When data is reused, processing may include manual steps to ensure format and coding compatibility, ensuring integration and suitability for the project.

- *Analyse*: Data analysis involves delving into the gathered data to comprehend the insights it holds and/or applying mathematical formulas (or models) to unveil relationships among variables.

- *Preserve*: Data is stored in a suitable long-term archive, such as a data center.

- *Share*: Data sharing involves making your data accessible to others, either within collaborative projects or through publication, with varying levels of access.

- *Reuse*: Data reuse involves repurposing data for new analyses, promoting independent research and fostering collaboration, aligning with Findable, Accessible, Interoperable, and Reusable (FAIR) principles for enhanced accessibility and integration.

The methodology employed in this study aligns with the earlier DLC. The following actions were undertaken for each step of the aforementioned DLC:

- *Planning*: It was developed following a line of work proposed in the reference [5], where the authors proposed the following: *"Implementation of a use case comparing different data decentralization architectures, such as the four proposed together with the Federated Learning approach: all reduce, ring all reduce, gossip and neighbor architecture."*[5]. Additionally, the current work continues a previous own study [1] where different openly available datasets were studied with data analysis techniques. One of these datasets was selected to board the actual study.

- *Collection*: Data was reused from a study of Detrano *et al.* [6].

- *Process*: Data was processed in order to have the best format to perform the distributed learning approaches. For this purpose, raw data from the original hospitals [10] was retrieved from an online digital library called the "Internet Archive", which stores copies of previous versions of web pages.

- *Analyze*: As mentioned before, distributed learning approaches will be applied with a focus on privacy preserving.

[1] `https://github.com/mma735/Practice-DS`

- *Preserve*: Data is stored in the GitHub repository: `https://github.com/mma735/TFM-DS.git`, which is suitable for long-term archive, particularly for code and smaller datasets. Additionally, the dataset is archived in the Zenodo repository, ensuring robust preservation policies. The Zenodo archive can be accessed here: `https://doi.org/10.5281/zenodo.10671543`.

- *Share*: Developed tools, input, and output data are openly accessible on the GitHub repository, licensed under the Apache 2.0 License.

- *Reuse*: Raw, processed and output data from models are ready to be reused in the GitHub repository.

In addition, we have started by collecting, processing, and curating data, focusing on descriptive statistics, missing value imputation, encoding of categorical variables, and data preparation for ML analysis, including train-test splitting and scaling. An important intermediate step involves examining data correlations and their implications. For a detailed visual representation of these steps, refer to the explanatory diagram provided in the Appendix A.

Concerning the analysis, the methodology followed in this study encompasses the following steps:

1. Designing an optimal ANN architecture, employing grid search with cross-validation.

2. Setting up clients for distributed learning.

3. Executing the five methodologies across numerous rounds of model weights sharing and model updating.

4. Identifying the optimal round based on classification metrics.

5. Comparing the performance evolution of the five methodologies.

6. Drawing conclusions based on the comparative analysis.

After data handling steps, an optimal ANN architecture suitable for the dataset is identified. This architecture is then utilized across the five distributed learning approaches. The data is divided among different clients, in this case, hospitals. The performance evolution of these decentralized approaches is compared by calculating metrics like loss, Area Under the Curve (AUC), and accuracy. Also, the examination of metric evolution for analyzing client communication is enhanced by incorporating a function known as weight divergence.

1.3.2 Contributions and structure of the work

Main contributions

FL operates with a structure that enables communication between all clients and a central server in each round, contrasting with the one-to-one or two-to-one client-to-client communication in other distributed architectures examined here. As a result, FL is poised to exhibit superior metric performance due to this collective communication approach, which is why it is prioritized for development.

However, exploring communication patterns between clients is also of interest (for example to reduce communication costs). To address this, the study also integrates RAR, GL, and a customized architecture. While these methodologies may offer potentially lower metric performance efficiency, they allow for a thorough examination of communication efficiency between pairs of clients. This examination is conducted through the analysis of individual metric evolution and weights divergences, aspects not feasible to explore within FL but critical in other architectures.

The detailed analysis of individualized metric graphs and divergence provides a deeper understanding of each client's role and efficiency in the distributed learning process.

Structure of the work

The subsequent Chapter 2, *"Theoretical basis and state of the art"*, begins by including the *"Theoretical basis"*, which is necessary for effectively understanding the technical terms used in this work. Then, the *"State of the art"* addresses distributed learning techniques in the context of privacy-preserving ML.

Chapter 3, *"Methodology and development"*, comprises a complete description of the use case dataset, conducts an exploratory analysis, presents the data cleaning process. Furthermore, it describes the experimental design for accomplishing distributed learning, searching for an optimal ANN architecture, and developing the learning techniques. Additionally, it describes the hardware and software used to allow replication of the results of this work.

In Chapter 4, *"Results and discussion"*, the performance of the different architectures is evaluated through classification metrics and an equation that measures the divergence between the clients' models is proposed and studied.

Finally, Chapter 5, presents the *"Conclusions"* drawn from the objectives outlined in Section 1.3. Additionally, this chapter discusses future work and research directions proposed as a continuation of this study.

Chapter 2

Theoretical basis and state of the art

2.1 Theoretical basis

Machine Learning represents a type of artificial intelligence that enables computer systems to autonomously learn and enhance their performance through data-driven applications. This involves the creation of algorithms and statistical models capable of analyzing data, recognizing patterns, and making informed predictions or decisions based on the acquired knowledge. Its applications span a diverse range, encompassing tasks such as image and speech recognition, fraud detection and recommendation systems among other. In addition, it is gaining prominence in sectors like finance, healthcare and transportation [11].

Supervised learning, for instance, involves training a model using labeled data where the target variable is known. This process equips the model to map input features to their corresponding outputs, thereby enhancing its ability to make accurate predictions on unseen data. On the other hand, *unsupervised learning* diverges from this path by focusing on unlabeled data, without any predefined target variable. In such cases, the model's objective is to identify inherent patterns and relationships within the data, a capability crucial for clustering, anomaly detection, and other exploratory data analysis tasks [11].

Distinct from both supervised and unsupervised learning is the realm of *reinforcement learning*. This paradigm shifts the focus to an interactive learning process, where an agent learns to make decisions by performing actions within an environment. Unlike the straightforward mapping of inputs to outputs in supervised learning, or the pattern discovery in unsupervised learning, reinforcement learning is characterized by a trial-and-error approach. The agent, through its interactions, receives feedback in the form of rewards or penalties. This feedback is fundamental in guiding the agent to learn optimal strategies over time [11]. Such a learning mechanism is particularly effective in complex decision-making scenarios, including game playing, robotics, and navigating dynamically changing environments [12].

A pivotal concept in modern machine learning is *Deep Learning*, which refers to artificial neural networks with multiple layers. These networks excel in handling large amounts of data and are instrumental in tasks like image and speech recognition, natural language processing and more [13].

In the context of *model optimization*, a crucial approach is *grid search*, wherein a thorough exploration of a predefined parameter space is conducted to pinpoint the most effective hyperparameters for a given machine learning model. This method, which usually integrates cross-validation, evaluates the model's performance on unseen data, thereby minimizing the risk of overfitting to the training dataset.

Cross-Validation (CV) is a valuable technique employed to evaluate a model's performance on new data. It achieves this by partitioning the available data into a training set and a validation one. The model is training on the designated training set, and its efficacy is subsequently evaluated on the validation set. A common variant of cross-validation is k-fold cross-validation, where the data is partitioned into k equally sized subsets, utilizing each subset iteratively as the validation set while the remaining $k - 1$ subsets form the training set.

To assess the performance of a *classification model*, various metrics can be utilized, such as *accuracy*, *precision*, *recall*, *f1-score*, *Receiver Operating Characteristic (ROC)*, and *area under the curve ROC (AUCROC or simply AUC)* [14].

- **Accuracy:** Represents the ratio of correctly classified instances to the total number of instances.
- **Precision:** Signifies the proportion of correctly predicted positive instances (true positives) over the total number of instances predicted as positive (true positives + false positives) for an specific class. Precision gauges the reliability of the results.
- **Recall:** For a given class, it measures the proportion of correctly classified positive instances over the total number of actual positive instances. Recall indicates how effectively the model identifies this class.
- **F1-score:** Is defined as the harmonic mean of precision and recall for a given class, offering a balanced evaluation of a model's performance.
- **ROC curve:** A graphical depiction of a binary classifier system's performance as the discrimination threshold varies. It plots the true positive rate against the false positive rate at different threshold settings. The optimal operating point is where the curve balances the two rates most appropriately for the specific problem.
- **Area Under the Curve ROC:** Measures the model's ability to differentiate between positive and negative instances by calculating the area under the ROC curve.

A steeper ROC curve signifies superior performance, as the classifier achieves high true positive rates while maintaining a low false positive rate.

Furthermore, a loss function assesses the performance of a model in making predictions, especially in problem-solving contexts. When the model's predictions closely align with the actual values, the loss is minimized. Conversely, if the predictions deviate significantly from the original values, the loss value increases. Specifically, in binary classification tasks (0 and 1), one commonly used loss function is the *binary cross-entropy loss.*

The formula for binary cross-entropy loss is as follows:

$$\mathcal{L}(y, p) = -\frac{1}{N} \sum_{i=1}^{N} (y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)) \tag{2.1}$$

where y_i is the true label for the i-th sample (0 or 1), p_i is the probability of class 1, $(1 - p_i)$ is the probability of class 0 and N is the total number of samples.

When data is missing, it can skew analysis outcomes as some observations might be omitted, resulting in incomplete or misleading findings. Moreover, managing and interpreting the data becomes more challenging, requiring appropriate handling of *missing values* through *imputation* or removal. Furthermore, the absence of information can hamper the effectiveness of machine learning algorithms, especially those dependent on comprehensive data for detecting data patterns and relationships [15].

Mean/median imputation is a technique used to fill in missing values by replacing them with the mean or median value of the entire feature column. If the median value closely resembles the mean, indicating a symmetric distribution, mean imputation is employed to maintain the central tendency of the data. However, if the distribution is skewed and the mean significantly differs from the median, median imputation is used to better represent the central tendency [16].

Applying mean/median imputation to the entire dataset might influence the distinct patterns within each class of data points in a binary target variable. This is due to the fact that the mean/median value is influenced by the data distribution, and if one class's distribution markedly differs from the other, the mean/median value may exhibit bias towards that particular class.

To mitigate this concern, a solution could involve employing *conditional mean imputation*. Here, the mean is computed separately for each class, and missing values are imputed with the mean corresponding to their respective class. By doing so, the individual trends of each class are preserved, reducing the likelihood of introducing bias.

ANNs are computational models inspired by the human brain's neural networks. They consist of layers of interconnected nodes or neurons, where each node simulates a neuron's activation in response to an stimuli. ANNs are capable of learning complex patterns in data by adjusting the weights of connections between nodes during the training process. This learning ability makes ANNs highly effective for several applications, including image and speech recognition, natural language processing, and predictive analytics [17].

Gradient descent is the cornerstone optimization algorithm for training ANNs. It aims to minimize the loss function, which measures the difference between the predicted output of the network and the actual target values. The algorithm updates the weights of the network in the opposite direction of the loss function's gradient concerning the weights, hence, gradually reducing the loss [17].

Batch gradient descent involves the use of the entire training dataset to calculate the gradient of the loss function and update the network's weights in each iteration. This approach ensures a stable convergence to the minimum of the loss function but can be computationally expensive for large datasets. The concept of a *batch* in this context refers to the set of all training examples used in a single update of the model parameters. Despite its computational demands, batch processing benefits from the full dataset's information to make informed updates, leading to robust model performance [17].

Within the architecture of ANNs, various types of layers play pivotal roles in enhancing the network's learning capability and performance. Among these, *dense layers*, *dropout layers*, and *batch normalization layers* are particularly noteworthy [13].

Dense layers are the fundamental building blocks of ANNs, consisting of fully connected neurons where each neuron in a layer is connected to all neurons in the previous and next layers. These layers are instrumental in learning high-level patterns in data by performing complex matrix multiplications and applying non-linear transformations [13].

Dropout layers serve as a regularization technique to prevent overfitting in neural networks. By randomly setting a fraction of input units to 0 at each update during training, dropout layers reduce the reliance on any single neuron, encouraging the network to learn more robust features that are not dependent on specific paths within the network [13].

Batch normalization layers are used to normalize the inputs of each layer, meaning they adjust and scale the activations. By stabilizing the distribution of these inputs, batch normalization layers help in speeding up the training process, making the network more stable and efficient. They achieve this by reducing the internal covariate shift, which refers to the change in the distribution of network activations due to the update in layers during training [13].

Incorporating these layers into ANN architectures significantly impacts the network's learning process, enhancing its ability to generalize from training data to unseen data. Dense layers provide the computational power, dropout layers offer a mechanism to combat overfitting, and batch normalization layers ensure that the optimization landscape is smoother and more navigable.

2.2 Distributed machine learning techniques

Federated Learning FL has recently emerged as an increasingly popular solution proposed with the aim of performing the training of privacy-preserving ML or DL models between distinct clients or data owners. The primary concept underpinning federated learning involves the decentralized analysis of data, ensuring that client data remains secure and it is not transmitted to central servers. This approach addresses data privacy and latency issues but introduces challenges such as costly communication, system heterogeneity, statistical heterogeneity, and including some privacy concerns. Ensuring encrypted communication between servers and clients in federated learning real world scenarios is crucial for privacy and security [5].

Gossip Learning GL, another decentralized learning approach, is a variation of Federated Learning. It is characterized by an unstructured communication scheme among workers or clients. Unlike traditional FL, GL operates without the need for a central server. Instead, clients directly share updates of their models, and the aggregation occurs in a distributed manner. This approach not only maintains the fundamental advantages of Federated Learning in terms of model training and data privacy but also enhances robustness by leveraging a distributed structure in terms of model parameters [18]. While GL may occasionally be less efficient compared to centralized approaches, it excels in scalability and resilience. These qualities make it a practical choice in distributed learning scenarios where central coordination is either impractical or undesirable.

In a similar vein to *GL*, the *RAR* variation eliminates the need for a central parameter server, but utilizing a ring-structured network among nodes. This setup allows for direct communication and updates exchange between nodes, fostering efficient model synchronization and a consistent global model. Notably, the architecture is designed to be resilient against communication bottlenecks and single points of failure, although it primarily supports synchronous communication [4].

The three main architectures implemented, analyzed and compared in this study can be summarized as follows:

1. FL: FL decentralizes data analysis, keeping user data on local clients and using a client-server architecture for updating the ML/DL model. The server aggregates models from clients who train the model on their data. This method can be optimized using techniques like compression for updates [5].

2. GL: As a fully decentralized approach, GL does not require a client-server architecture. Nodes directly exchange and aggregate models. The communication between the clients could be done randomly or conditionally (fixed) to exchange updates during each iteration. This method is scalable and robust due to its decentralized nature, though it may sometimes be less efficient than other methods [18].

3. RAR: This approach does not feature a central client-server architecture. It involves a ring-structured network where each client communicates and updates models directly with adjacent ones. This structure facilitates efficient parallelization and synchronization of model updates across the network [4].

2.3 State of the art

Distributed machine learning, and particularly FL, has gained significant traction due to its capability to handle data decentralization while preserving privacy. This approach is especially relevant in fields like medical data analysis, cybersecurity, and Internet of Things applications. FL has been evolving since its introduction by McMahan in 2017 [19], focusing on collaborative and decentralized data analysis without the need for central data storage. This technique allows various users or devices, termed as clients, to contribute to the learning task of a model, coordinated by a central server. The key advantage here is that the client data never leaves its original location, only the model updates are shared, thus ensuring user's privacy [5]. In FL, challenges such as system heterogeneity, communication costs, data distribution disparities, and privacy concerns are significant.

Systems heterogeneity refers to the varying capabilities of devices participating in the learning process, some of which might be intermittent and not always available for training. Heterogeneity primarily results in different network bandwidth resources of the clients. Therefore, optimizing communication is crucial, which can be divided into two fronts: communication frequency optimization and communication costs optimization. The former avoids frequent update exchanges by controlling the communication rounds, while the latter reduces the size of updates in each communication round.

To address *communication frequency optimization*, McMahan *et al.* [4][19] introduced, in 2017, FL with their Federated Averaging (FedAvg) approach, which minimizes communication rounds by performing additional computations locally. Wang *et al.* [4][20] proposed in 2019 a dynamic control algorithm that adjusts the global aggregation frequency to minimize learning loss under fixed resource constraints.

Reisizadeh *et al.* [4][21] introduced in 2020 FedPAQ, a periodic average and quantization algorithm, allowing clients to update gradients locally and send quantized updates to the server. Quantized updates refer to a process where instead of sending the exact gradient updates calculated by each client, these updates are rounded or compressed into a smaller set of predefined values before being transmitted to the server. This approach reduces communication rounds and costs per round.

However, prior works perform *random client selection and that may result in unreliable selection and unnecessary bandwidth consumption.* AbdulRahman *et al.* [4][22] addressed this issue in 2020 by maximizing user participation in FL training and reducing discarded communication rounds through comprehensive evaluation of user capabilities, including CPU, memory, energy, and time.

Regarding *communication costs optimization*, Lin *et al.* [4][23] identified in 2017 redundant gradient exchanges in distributed stochastic gradient descent, leading to the adoption of techniques like gradient compression. Their Deep Gradient Compression (DGC) approach maintains accuracy without extra normalization layers by utilizing momentum correction and local gradient clipping. Caldas *et al.* [4][24] proposed in 2018 Federated Dropout, inspired by the dropout regularization technique, enabling local training of sub-models that contribute to the global model. Chen *et al.* [4][25] introduced in 2019 an asynchronous learning strategy aimed at reducing communication costs between the server and clients. This strategy involves dividing the ANN layers into shallow and deep layers. Shallow layer parameters are updated more frequently than deep layer parameters, further decreasing communication overhead.

While FL offers *privacy protection* for users, recent studies have revealed vulnerabilities that allow for the extraction of user information from trained models. For instance, Hitaj *et al.* [4][26] devised in 2017 a distributed deep learning attack leveraging Generative Adversarial Networks (GAN). This attack enables any participant in collaborative training to generate sensitive data about the target device. Interestingly, the attack remains effective even if the client employs differential privacy techniques to obscure parameters. The authors argue that this attack vector is similarly applicable to FL scenarios.

In FL with client-server architecture, privacy threats arise from two primary sources: the client-server architecture and malicious edge nodes. Initial studies primarily addressed privacy concerns from the client-server architecture, employing techniques such as differential privacy, homomorphic encryption, and secure multi-party computation. Subsequent research introduced blockchain technology to counter privacy threats from the client-server architecture and partially mitigate risks from malicious nodes or users.

Li *et al.* [4][27] concluded in 2019 that *data distribution disparities* between distinct clients slows down the convergence rate in the FL approach. Also, Duan *et al.* [4][28] demonstrated that imbalanced distributed training data have the potential to diminish the accuracy of a model trained using FL. The FedAvg algorithm proposed by McMahan *et al.* [4][19] is proved to be resilient to a certain degree of data distribution disparities, but recent findings, in 2018, by Zhao *et al.* [4][29] indicate a significant decrease in model accuracy when the FedAvg algorithm confronted with datasets with highly disparities.

Yoshida *et al.* [4][30] introduced, in 2020, Hybrid-FL, a novel learning approach aimed at addressing the data distribution disparities. They proposed uploading data from select clients to the server to create an approximately identically distributed dataset, which was then used for model training. This trained model was combined with others trained on data with clear distribution disparity.

Also, the global data imbalance could be mitigated with data augmentation techniques, as proved by Duan *et al.* [4][28]. To tackle local data distribution imbalances, they introduced a mediator that reorganizes client training based on the Kullback-Leibler divergence of clients' data distributions. However, this approach may inadvertently disclose client privacy due to data distribution analysis.

While FL with client-server architecture provides a degree of user privacy protection, it suffers from several limitations. Firstly, it creates communication bottlenecks, and secondly, it relies heavily on the client-server, requiring the unanimous selection of a trusted server by all clients. Additionally, server failures can disrupt the entire training process. To address these issues, some studies have proposed discarding leveraging technologies such as blockchain and alternative communication methods [4].

For instance, Lalitha *et al.* [4][31] devised in 2019 a decentralized FL approach where all nodes on a network communicated with their neighbors to train the model, known as the *neighbor architecture*.

Hegedűs *et al.* [4][32] conducted in 2020 a systematic comparison of aggregation costs between FL architecture and GL architecture, finding comparable performance between the two.

Hu *et al.* [4][33] segmented, in 2019, the model into non-overlapping segments, each node aggregating local segments and those from other nodes before randomly transmitting them to a certain number of other nodes. This approach significantly reduces communication costs compared to transmitting the entire model, and it is known as *Split Learning*.

As already explained, random GL is other alternative to FL, where communication between clients are randomly selected, but same as the previous mentioned works of FL which performs random client selection, some of the issues related to random client selection still persist [18]:

- Heterogeneity of clients: Even without a central server, clients may still vary in terms of their computational resources, network connectivity, and data distributions. Randomly selecting clients for communication without considering these factors can lead to inefficient utilization of resources and sub-optimal learning outcomes.

- Data imbalance: Random selection may still result in an imbalance in the distribution of data across clients. Some clients may possess more relevant or representative data for the learning task than others. If important data sources are consistently excluded or underrepresented due to random selection, the resulting model may be biased or less accurate.

- Bandwidth consumption: Randomly selecting clients for communication may lead to inefficient use of bandwidth, especially if clients with limited bandwidth or unstable connections are included in the selection process. This can result in communication delays, packet loss, and increased network congestion, ultimately affecting the overall performance of the distributed learning system.

Chapter 3

Methodology and development

This chapter provides a detailed overview of the procedural steps undertaken in this study. It begins by outlining the data sources used, followed by a comprehensive description of the dataset. Subsequently, the data is subjected to rigorous processing and curation procedures to ensure its quality and suitability. Next, the dataset is analyzed to study and select the predictor variables, as well as segmented by clients focusing on the objective variable.

Following these initial steps, the chapter delves into the design of the experiment involving distributed learning approaches. Initially, it outlines the design of the ANN architecture, which serves as initial model for the distributed approaches developed.

3.1 Data Collection and Preparation

In the realm of distributed ML/DL, the motivation for utilizing openly available data, such as the "Heart Disease" dataset using in this study, stems from the desire to simulate a potential real-world scenario. In this scenario, multiple hospitals aim to collaborate on constructing a comprehensive Artificial Intelligence (AI) (ML/DL) model to advance healthcare outcomes. However, they may face significant constraints regarding data sharing among themselves or with a central server due to stringent data privacy regulations. This situation underscores the necessity of employing data that, while publicly accessible, mirrors the complexities and challenges of dealing with sensitive medical information.

Medical data, inherently sensitive due to its personal nature, demands meticulous attention to privacy concerns. The utilization of such data in this study is guided by a dual objective: to demonstrate the feasibility of collaborative model building under privacy constraints and to highlight the effectiveness of distributed learning techniques in managing sensitive information securely. Through this approach, the research not only adheres to privacy-preserving principles but also contributes to the broader discourse on secure and efficient data handling in medical research.

3.1.1 Data Source

The so-called "Heart Disease" dataset [34] comes from the combination of different datasets obtained from hospitals in different countries. It was created by consolidating 5 datasets based on 11 shared features. This combined dataset is currently the most extensive resource for studying heart disease and the source is described in Table 3.1.

In the table the first column shows the abbreviation used in the subsequent procedural steps and the quantity of data n. The second column shows the hospital of origin of the data and the author.

Study group	**Hospital and author**
"hung" $n = 293$	Hungarian Institute of Cardiology, Budapest Andras Janosi, M.D.
"swit" $n = 184$	University Hospital (Zurich and Basel), Switzerland William Steinbrunn, M.D. and Matthias Pfisterer, M.D.
"long" $n = 138$	V.A. Medical Center, Long Beach Robert Detrano, M.D., Ph.D.
"stat" $n = 151$	Statlog dataset (confidential source) King, Ross D., Ph.D.
"clev" $n = 151$	Cleveland Clinic Foundation Robert Detrano, M.D., Ph.D.

Table 3.1: Description of the distribution of patient data with heart disease.

All of these datasets mentioned can be found under the "Index of heart disease datasets" from a previous version of a UCI Machine Learning Repository [10] and was first collected for a study of Detrano *et al.* called "International Application of a New Probability Algorithm for the Diagnosis of Coronary Artery Disease" [6].

3.1.2 Description of the dataset

This merged dataset contains in total 918 observations with 12 attributes, 11 medical predictor variables of numerical and categorical type and the target variable "Heart Disease". The input and output variables are characterized below:

- *Age*: Age (years), numerical variable.
- *Sex*: Gender; categorical variable (M, F).
- *ChestPainType*: Chest pain type; categorical variable (TA: Typical Angina, ATA: Atypical Angina, NAP: Non-Anginal Pain, ASY: Asymptomatic).
- *RestingBP*: Resting blood pressure; numerical variable (mm Hg).
- *Cholesterol*: Serum cholesterol; numerical variable (mm/dL).
- *FastingBS*: Fasting blood sugar; categorical variable (1: if FastingBS $>$ 120 mg/dL, 0: otherwise).
- *RestingECG*: Resting electrocardiogram results (Normal: Normal, ST: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of $>$ 0.05 mV), LVH: Showing probable or definite left ventricular hypertrophy by Estes' criteria); categorical variable.
- *MaxHR*: Maximum heart rate achieved; numerical variable.
- *ExerciseAngina*: Exercise-induced angina; categorical variable (Y: Yes, N: No).

- *Oldpeak*: Represents the ST depression (mm) induced by exercise relative to rest; numerical variable.
- *ST_Slope*: The slope of the peak exercise ST segment (Up: upsloping, Flat: flat, Down: downsloping); categorical variable.
- *HeartDisease*: Target variable (1: heart disease, 0: Normal).

3.1.3 Data Processing and curation

Data cleaning involves searching for and deleting duplication values, fixing categorization errors, outliers removal, exploring, and imputing missing data. All these steps are summarized in the diagram provided in the Appendix section (see Appendix A, "Data processing and curation" steps in green color).

Outliers in the "Heart Disease" dataset could be crucial for diagnosis, as sometimes individuals with this disease may exhibit variables with these values. Their study will be conducted in the next Section 3.1.4.

In addition, Not a Number (NaN) values were found in the form of illogical zero values. In this dataset, *Cholesterol* and *RestingBP* variables had zero values. Another variable, *Oldpeak*, contained many zero values, and without domain knowledge, determining if these values are NaN is challenging. As described in Subsection 3.1.2 and referenced in the bibliography [35], an ST segment measurement of 0 mm could be considered correct and is often observed, particularly in patients without heart disease. This is illustrated in the violin plots presented in Figure 3.1.

NaN values are important missing data for the study carried out, which must be imputed in order to preserve valuable information. Simple mean/median imputation methods aim to balance the distributions across classes, potentially sacrificing the unique trends within each class. This trend is crucial information that aids machine learning algorithms. Conditional mean imputation retains these distinctive trends, avoiding potential biases introduced by mean imputation across the entire dataset. So, this latter missing values imputation technique will be used.

Afterwards, the process involved data transformation, data splitting, and data scaling. Categorical variables were converted into numerical values using the OneHotEncoder module from the scikit-learn library [36]. This class generates binary columns for each category within the categorical feature. A value of 1 represents the presence of a category, while 0 signifies its absence. This conversion enables seamless handling by the ML algorithms. At this point, the number of predictor variables has changed from 11 to 20. This transition occurs because OneHotEncoder expands each categorical variable into multiple binary variables, one for each unique category. For instance, if a single categorical feature contains 10 unique categories, OneHotEncoding transforms this single feature into 10 separate binary features.

Following data transformation, the dataset was split into training and test sets, allocating 75% to training and the remaining 25% to testing, employing the train_test_split method [36].

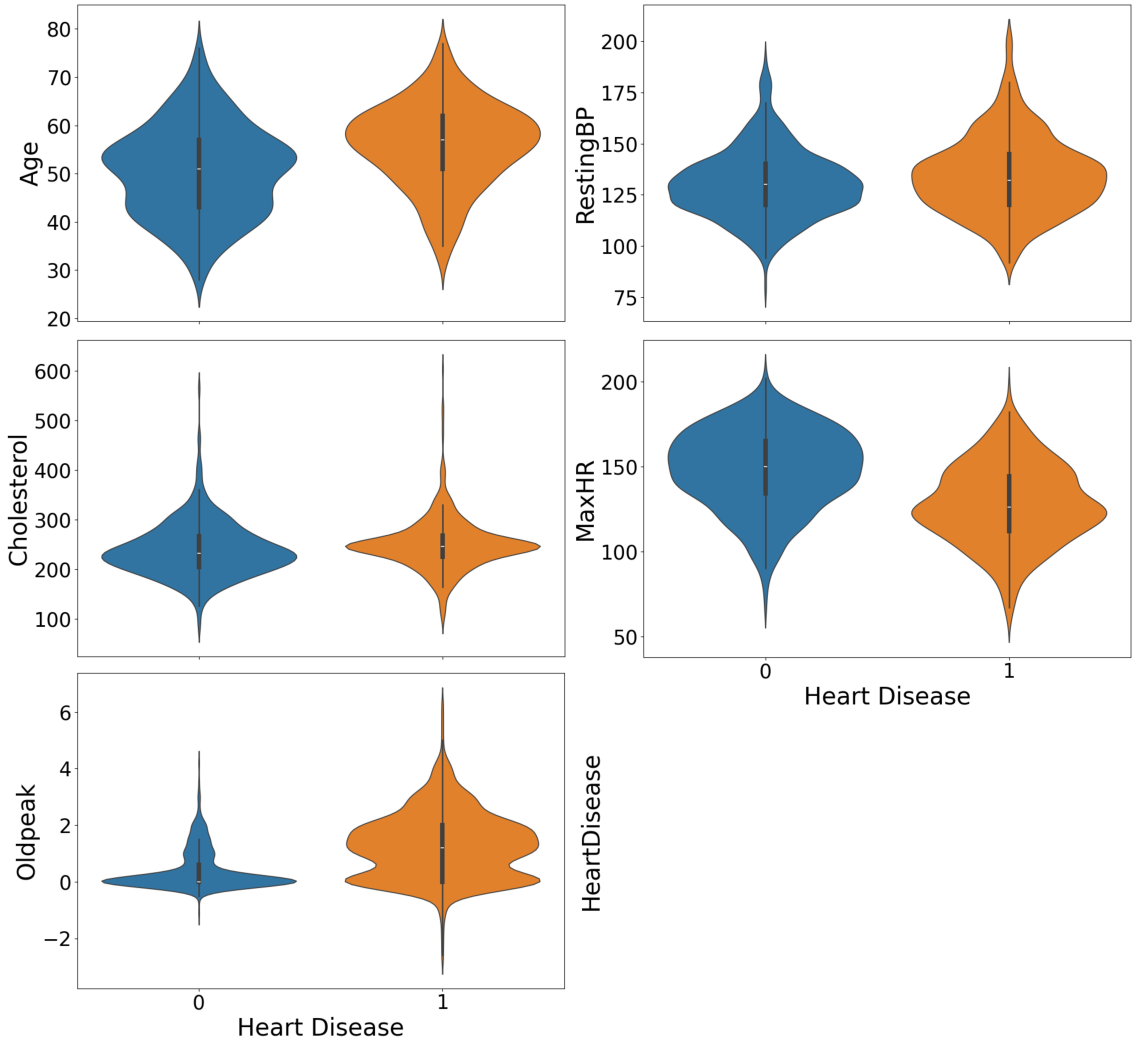


Figure 3.1: Visual representations of the numerical attributes within the "Heart Disease" dataset are depicted in violin plots, distinguishing data points into class 0 (people without heart disease) and class 1 (people with heart disease).

Subsequently, scaling was applied to ensure uniformity across all features, potentially enhancing the accuracy of the machine learning model. Utilizing the MinMaxScaler[36], scaling was exclusively fitted to the training data. The identical scaling parameters were then applied to both the training and test sets, mirroring real-world, unseen data scenarios [37].

3.1.4 Data analysis

Data analysis were performed both in the entire dataset to study predictor variables and the dataset divided by clients focusing in the objective variable.

Entire dataset

The Table 3.2 shows the *descriptive statistics* of the numerical features of the dataset. It can be seen that the data are spread out too. The zero value for *RestingBP* shown as minimum is impossible for a living human, also values higher than 150 mm Hg are considered hypertensive crisis and requires urgent medical attention. People with a value of zero for *Cholesterol* is also very rarely, so we consider that it has to be NaN values

which, probably, is the cause of the considerable difference between mean and median.

	count	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
Age	918	53.51	9.43	28	47	54	60	77
RestingBP	918	132.40	18.51	0	120	130	140	200
Cholesterol	918	198.80	109.38	0	173.25	223	267	603
MaxHR	918	136.81	25.46	60	120	138	156	202
Oldpeak	918	0.89	1.07	-2.6	0	0.6	1.5	6.2

Table 3.2: Descriptive statistics of the numerical variables for Heart Disease dataset.

Table 3.3 shows the percentages of outliers. In this analysis, an outlier is defined as a data point that falls outside the interquartile range. As previously mentioned, *Cholesterol* feature don't have outliers because there are NaN values in the form of zero values. The other features shows no significant quantity of outliers and, as mentioned before, they can be crucial data for prediction purposes.

Variable	**Outliers (%)**
RestingBP	3.05
Cholesterol	19.96
MaxHR	0.22
Oldpeak	1.74

Table 3.3: Percentages of outliers for each variable of the Heart Disease dataset.

A *correlation analysis* was conducted using the Spearman method to account for outliers and unknown variable relationships, making it preferable over Pearson correlation. Assessing the correlations between predictor and target variables helps to reveal the strength and direction of their relationship, although correlation does not imply causation. However, in the medical domain, it's understood that the variables in this study significantly influence heart disease.

The correlation matrices for the processed data of the "Heart Disease" dataset are depicted in Figure 3.2. Here we can note that predictor variables such as *ST_Slope_Flat*, *ChestPainType_ASY*, *ExerciseAngina_Y*, *Oldpeak*, and *Sex_M* exhibit moderate to strong positive correlations (0.3 - 0.6) regarding the target variable. These variables are anticipated to be reliable predictors. Moreover, numerous pairs of numerical predictors that were categorical before one-hot encoding demonstrate high correlations in this dataset.

In principle, addressing multicollinearity may not be a significant concern when applying ANN models, as they can effectively handle correlated features compared to conventional ML models.

Dataset divided by clients

It is important to control the balance of data in binary classification problems and, in this case, of each distinct client. The analysis of this issue it is summarized in Tab. 3.4. The entire dataset is balanced, but special attention should be paid to future clients "long" and "swit". In what follows, the clients "clev", "hung", "long", "stat", and "swit" will be referred to interchangeably as client 1, client 2, client 3, client 4, and client 5 respectively.

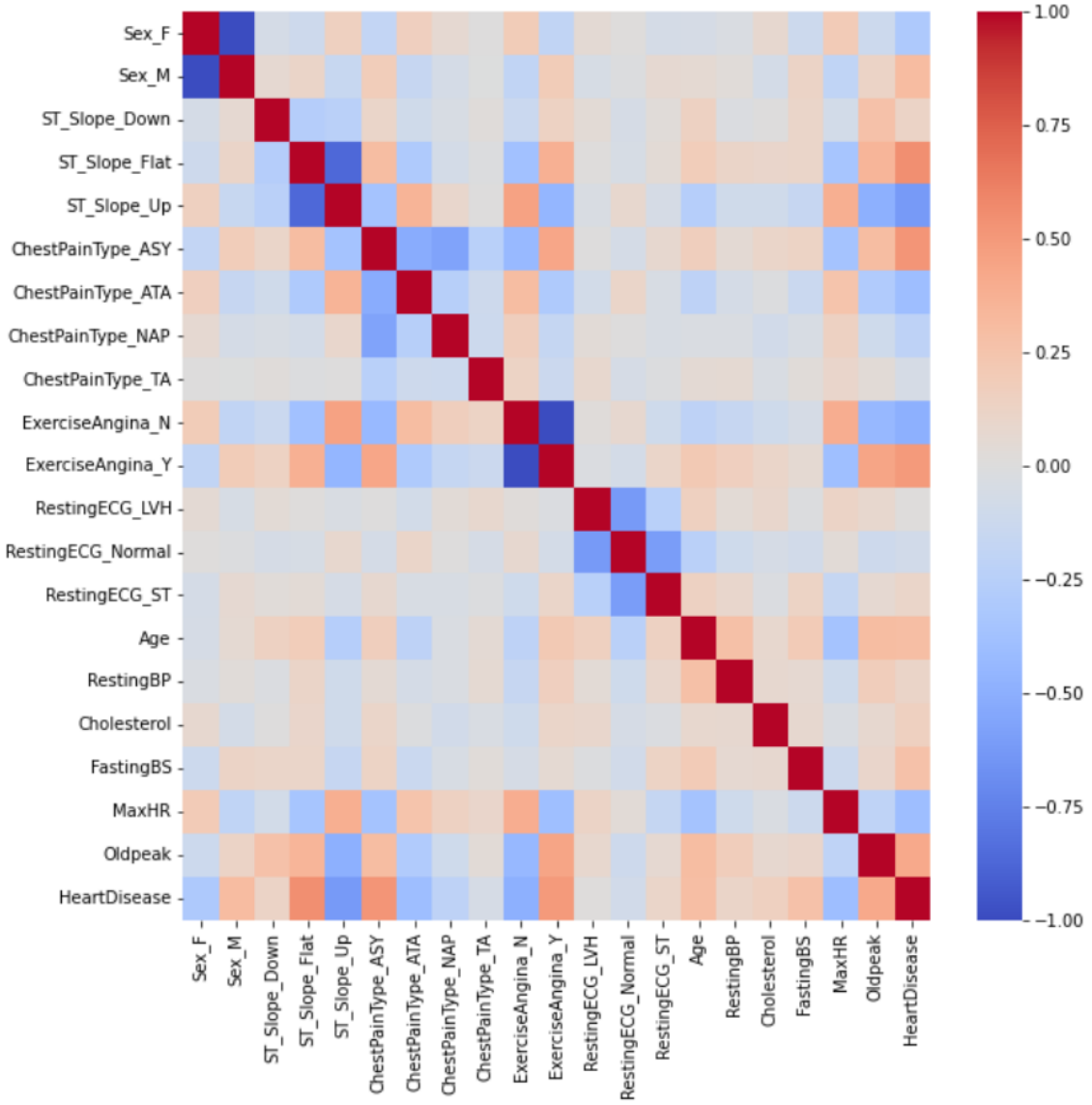


Figure 3.2: Spearman correlation matrix for the variables of the Heart Disease dataset.

Cases that do not reflect the general statistics may be due to factors such as the hospital collecting data after making a prior diagnosis or conducting screening, for example, of an older age group, as is the case with the V.A. Medical Center in Long Beach, which is a hospital that provides care to U.S. military veterans.

For the purpose of the present work, because the distribution of data among different clients is not homogeneous and does not follow a pattern of statistical independence, there could be *implications for the performance and convergence of the simulated distributed learning architectures.*

3.2 Experimental Design

The main objective of the present work is to compare the performance of different distributed learning techniques. It is worth remembering that here we are dealing with distributed learning from the point of view of privacy, that means that original data remains locally and only model parameters are shared, this minimizes the risk of exposing personal or sensitive data. Then, *this approaches are used to preserve data privacy and*

Name	% Positive Cases	% Negative Cases	Total Cases
clev	45.4	54.6	152
hung	36.2	63.8	293
long	76.1	23.9	138
stat	46.4	53.6	151
swit	85.9	14.1	184
Total	**55.3**	**44.7**	**918**

Table 3.4: Percentage of Positive and Negative Cases by Hospital

comply related regulations.

To carry out the above, we started by designing an ANN model, that will be common to all clients in all architectures at the beginning of the first round.

Once the distributed learning was designed taking into account privacy, the architectures were run and the performance was evaluated with classification metrics such as loss, accuracy and AUC.

The observation of the metric evolution to study the communication between clients is complemented by a modified function of the one defined as weight divergence, $d_{m,n}(t)$, as shown in Equation 3.1.

$$d_{mn}(t) = \frac{\|w_m(t) - w_n(t)\|}{\frac{1}{2}(\|w_m(t)\| + \|w_n(t)\|)} \tag{3.1}$$

Here, $w_m(t)$ and $w_n(t)$ represent the weights of clients m and n respectively at round t.

The implemented code in Python for weight divergence calculation is shown in the Code Block 3.1. The function receives model weights for two clients and returns the divergence value. Internally, first it flattens the arrays of weights into one dimension to performs calculation of subtraction or norm calculation given the Equation 3.1.

```
def weight_divergence(weights_m, weights_n):
    """
    Compute the weight divergence between two sets of weights.

    :param list weights_m: Set of weights for model m
    :param list weights_n: Set of weights for model n
    :return divergence: Weight divergence between the two sets of weights
    """
    # Flatten the weights arrays to 1-dimensional arrays
    flat_weights_m = np.concatenate([w.ravel() for w in weights_m])
    flat_weights_n = np.concatenate([w.ravel() for w in weights_n])

    # Compute the Euclidean norm of the difference between
    # weights_m and weights_n
    norm_diff = np.linalg.norm(flat_weights_m - flat_weights_n)

    # Compute the Euclidean norm of weights_m and weights_n
    norm_m = np.linalg.norm(flat_weights_m)
    norm_n = np.linalg.norm(flat_weights_n)
```

```
    # Calculate the weight divergence
    divergence = norm_diff / (0.5 * (norm_m + norm_n))

    return divergence
```

Code Block 3.1: Python code implementation of weight divergence calculation.

This equation measures the weight divergence between client models, which is crucial for understanding how model weights differ across various clients and rounds of the distributed algorithm. Most importantly, it allows us to discuss the effect of the distributed learning technique under study.

It should be mentioned that this equation is an adaptation of the concept of divergence defined by Zhang *et al.* [38]. The authors used the equation dividing by the norm of $w_n(t)$ instead of by the mean of the norms of both $w_n(t)$ and $w_m(t)$, as it is used in this work. This is because the study of Zhang *et al.* only applied the equation to a unique baseline "c0" to the 100 clients studied [38], whereas here the interest lies in comparing all pairs of clients' weights. *Given that a symmetric metric is desirable, this adaptation was made.*

3.2.1 Design of the ANN architecture

In a real-world scenario, the dataset with the most data from the client would be selected to establish this optimal ANN architecture because merging the data for all clients would be incompatible with privacy concerns. Applying this to the "Heart Disease" dataset, would lead us to select the "hung" client to determine the optimal ANN architecture. But, in order to achieve optimal distributed learning results, it is possible to apply the search for the optimal ANN architecture using the medical data from all clients. This avoids the need to separate a validation set (a 20% or 30% of the training data) for each client when conducting distributed learning, providing more data for training these distributed architectures.

Previous splitting the client data into 75%/25% train/test subsets and transforming it with data scaling (MinMaxScaler), it was performed GridSearchCV for various distribution of neurons. GridSearchCV performed hyperparameter tuning through grid search and CV on the training data. The number of k-folds was set to 3, and the implementation was carried out using the Python library Keras.

The grid was composed of the following values:

- "batch_size": [8, 12, 14, 16, 18, 20, 22],
- "epochs": [25, 31, 38, 44, 50, 75, 85, 100]

This search resulted in the ANN architectures shown below:

- **Dense layer.** Units 256. Activation: *ReLU*. Regularization function: L1 and L2 with a parameter of 0.01. Input shape: (20,).
- **BatchNormalization layer.**
- **Dropout layer.** Rate: 0.5.
- **Dense layer.** Units 256. Activation: *ReLU*.

- **BatchNormalization layer.**
- **Dropout layer.** Rate: 0.5.
- **Dense layer.** Units 256. Activation: *ReLU*.
- **BatchNormalization layer.**
- **Dropout layer.** Rate: 0.5.
- **Dense layer.** Units 128. Activation: *ReLU*.
- **BatchNormalization layer.**
- **Dropout layer.** Rate: 0.5.
- **Dense layer.** Units 32. Activation: *ReLU*.
- **BatchNormalization layer.**
- **Dropout layer.** Rate: 0.5.
- **Dense layer.** Units 1. Activation: *sigmoid*.

Also, a graphical representation of the designed architecture is shown in the Appendix B for better understanding of the model.

The grid search CV identified the optimal values as "batch_size": 14 and "epochs": 38. Following the division of the entire "Heart Disease dataset" into a 0.75%/0.25% train/test split, the calculation of test metrics yielded a loss of 0.72, an accuracy of 0.8, and an AUC of 0.89. These results represent the highest performance achieved, thereby fulfilling the criteria for selection as the optimal hyperparameters. Further detail are available in the GitHub repository: `https://github.com/mma735/TFM-DS.git`.

The models were trained using the *Adam* optimizer with a learning rate set to 0.001, and the binary cross-entropy function served as the loss metric. The models' performance was evaluated based on metrics such as AUC and accuracy.

3.2.2 Distributed Machine Learning Techniques

Having found the optimal architectures, the next step is to run the distributed learning architectures, incorporating on each client, once again, data scaling with MinMaxScaler [36] to enhance the convergence and interpretation of the ANN.

All of the designed distributed approaches consist of repeating cycles of communication schemes, where the ANN model of each client is updated, improving its performance.

Federated Learning approach

The first designed approach is FL. The cycle begins with the server providing the same model, which is then trained by the clients using their respective data. Only the weights generated by each client are sent to the server, which aggregates these weights and updates the model. This communication process is repeated as many times as necessary. In this work, the number of rounds N_r is fixed at 100, deemed sufficient for the model to converge to a feasible result.

The scheme followed by the FL architecture is depicted in Figure 3.3, labeled with numbers corresponding to the following steps [5]:

1. **Server:** generates the model for local training by each model.
2. **Server:** sends the model to all of the clients.
3. **Client:** trains each model received with its own data. Ideally, a portion of the initial data should be set aside for validation and testing of the model's performance using classification metrics. However, in this work, the validation part is omitted as explained in Section 3.3.
4. **Client:** sends the model parameters to the server, not the data. In a real-world scenario, this communication channel could be protected with encryption techniques.
5. **Server:** receives all the model weights, applies an aggregation function, for example, the weighted average depending on the number of data from each client, and updates the model.
6. Iterate the procedure from step 2 to initiate new communication rounds with the new updated model.

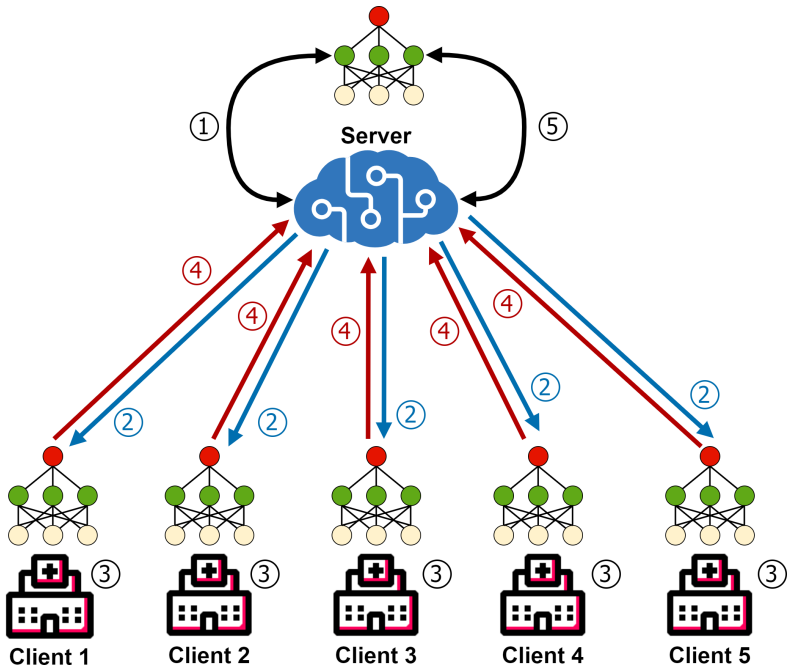


Figure 3.3: Scheme of a Federated Learning architecture implemented. Adapted from [5].

As the server updates the model with the weights or parameters of *all the clients for each round*, in some cases the FL approach can be faster than the other architectures mentioned in Section 2.1, where communication is less effective. Nevertheless, the rest can allow to study the client-to-client communication. This last is also of great importance in the present work, being to reason to explore these architectures, including other customized one.

The Code Block 3.2 shows a simulated FL architecture implementation in Python code. It performs a "for loop" a number of times given to simulate the communication rounds. Inside each round, the code starts by training the ANN model of each client with

other "for loop" running a dictionary.

Then, the model weights of all the clients are aggregated with an aggregation function (see Code Block 3.3) that takes into account the number data points of each client to perform a weighted average. Finally, the weights of the aggregated model of the round are saved in a list, ready to be used to calculate the classification metrics in test.

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

    :param n_times: Number of communication rounds
    :param initial_weights: Initial weights from optimal ANN model
    :param n_i: List with number of samples for each client
    :param model: optimal ANN model
    :param data_dict: Dictionary of client data with train and test split
    :return save_weights: Aggregated model weights of all the rounds
    """
    # To save client weights of all the rounds
    save_weights = []
    # Perform communication rounds
    for i in range(1,n_times+1):

        # Collect client weights of a round
        weights_round = list()

        # Generate weights for each client
        for name in data_dict.keys():
            # Setting the initial weigths to client
            model.set_weights(initial_weights)
            # Train client:
            X_train_np = np.array(data_dict[name]['X_train'])
            y_train_np = np.array(data_dict[name]['y_train'])
            # Training with optimal parameters found in the design of
            # the ANN architecture
            model.fit(X_train_np, y_train_np, epochs=38, batch_size=14)
            # Obtain client weights:
            weights_client = model.get_weights()
            # Save weights of each client:
            weights_round.append(weights_client)

        # Aggregate the weights obtained with each client
        avg_weights = ave_weights(n_i,weights_round)
        # Set the aggregated weights to model
        model.set_weights(avg_weights)
        # Initial weights for the next round
        initial_weights = avg_weights

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

Code Block 3.2: Simulated FL architecture algorithm in Python code.

The aggregation function, shown in Code Block 3.3, performs a weighted average depending on the number of data from each client, as mentioned before. In FL, it aggregates the model weights of all the clients, but it is also used in other client-to-client architectures to perform aggregation with the model parameters of only selected clients. If the architecture involves one-to-one communication, only two client weights are added to the aggregation function. However, if it involves two-to-one communication, three client weights are added to this function.

```
def ave_weights(n_i,listOfWeights):
    """
    Aggregation function

   :param list n_i: Number of samples for each client
   :param list listOfWeights: Weights for each client
   :return: Final weighted average for global model
    """
    N = sum(n_i) # total number of samples of all clients
    # initial weights of global model, set to zero
    ave_weights = listOfWeights[0]
    ave_weights = [i * 0 for i in ave_weights]
    # loop whose range is number of clients
    for j in range(len(n_i)):
        # receive weights from clients
        rec_weight = listOfWeights[j]
        # multiply the client weights by number of local data samples
        # in client local data
        rec_weight =  [i * n_i[j] for i in rec_weight]
        # divide the weights by total number of samples of all clients
        rec_weight =  [i / N for i in rec_weight]
        # sum the weights of new client with the prior
        ave_weights = [x + y for x, y in zip(ave_weights,rec_weight)]
    return ave_weights
```

Code Block 3.3: Python code of an aggregation function that performs a weighted average of the ANN model parameters of the given clients.

Procedure in client-to-client architectures

The subsequent designed approaches are shown in Figure 3.4. The big difference is that they don't need a server to send the weights, they send it to other clients. They are described in the way they are developed in Python[1], it is created a list "clients" with the five clients on each position, each client send the weights to the same position of the list "clients_send". This configuration allows the possibility to add more than one client in a position of the "clients" list, meaning than these clients will send the model weights to a client of "clients_send". Once a client receives the model weights of others in a communication round, it is performed the aggregation with the weighted average to update the local model of the client. Again, the communication rounds are set to $N_r = 100$.

[1]See entire code in GitHub: `https://github.com/mma735/TFM-DS.git`

The communication architectures are described below, in terms of the "clients" and "clients_send" lists:

- **Ring All-Reduce:** It has the "clients" to "clients_send" design of [5,1,2,3,4] → [1, 2, 3, 4, 5]. In this way, Client 5 communicates its weights to 1, 1 to 2, 2 to 3, and so on, resulting in a ring structure where client N receives the weights of client $(N-1)$ if $N = 2, 3, 4, 5$ and client 5 if $N = 1$.

- **Conditional Gossip Learning:** It have the design [3,1,[2,5],3,4] → [1, 2, 3, 4, 5]. Here the big difference is that the third position of the "clients" list have two clients and then, the target client in the "clients_send" list performs a weighted average with three weights: two received from other clients and one it has from its previous model.

- **Customized architecture:** This architecture is designed so that the client N receives the weights of clients $N-1$ and $N-2$ if $N > 2$, if $N = 2$, it receives the weights of clients 1 and 5, and if $N = 1$, it receives the weights of clients 5 and 4. This results in the architecture sequence [[4,5],[5,1],[1,2],[2,3],[3,4]] → [1, 2, 3, 4, 5].

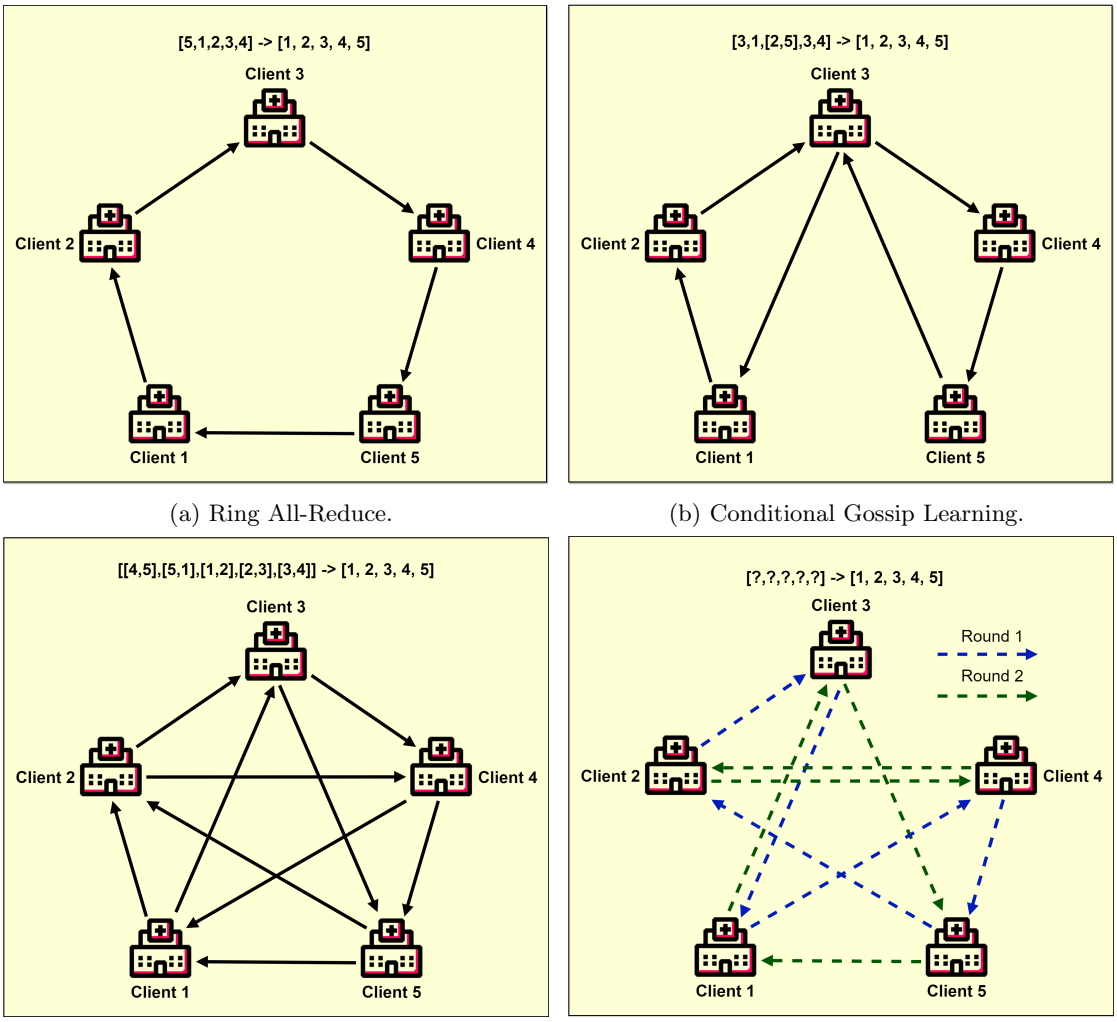
- **Random Gossip Learning:** Following the assignment of "clients" to "clients_send", it would have the design [?,?,?,?,?] → [1, 2, 3, 4, 5], randomly changing the list "clients" with the only restriction of not to having the same client in the same position in both lists (a client communicating with itself would not make sense). An example of communication for the first two rounds is shown in Figure 3.4.

To share and explain Code Blocks of all the client-to-client approaches would be quite large due to its complexity. Therefore, they are entirely shown step-by-step in the following GitHub repository: `https://github.com/mma735/TFM-DS.git`. RAR and random GL have a one-to-one communication and they have a similar and less complex implementation compared to conditional GL and the customized architecture, where it is found a case of two-to-one communication.

The RAR architecture implemented in Python code is shown in the Code Block 3.4.

There are two main differences with FL code (see Code Block 3.2). The first difference is that, after generating the model weights of each client, these parameters are introduced in an specific position of a list called "weights_clients" given the RAR architecture design: [5,1,2,3,4] → [1, 2, 3, 4, 5]. Then, "weights_clients" will have the following distribution of weights, where the first position corresponds with the first client, the second position with the second client and so on. Each, of these positions will have two model weights, the client's own weights and the model weights that receives from other client. So, "weights_clients" will have the following appearance:

```
weights_clients = [[weights_clients1, weights_clients5],
[weights_clients2, weights_clients1],
[weights_clients3, weights_clients2],
[weights_clients4, weights_clients3],
[weights_clients5, weights_clients4]]
```



(a) Ring All-Reduce.

(b) Conditional Gossip Learning.

(c) Customized architecture.

(d) Random Gossip Learning.

Figure 3.4: Schemes of client-to-client architectures of distributed learning implemented. Above each diagram, the communications between "clients" and "clients_send" are shown.

The second difference is observed in the addition of other "for loop" to perform the aggregation on each position of the "weights_clients" list. The aggregation results in this initial appearance:

```
weights_clients = [[weights_clients1],
[weights_clients2],
[weights_clients3],
[weights_clients4],
[weights_clients5]]
```

Now, the "weights_clients" list is ready to be used in the next communication round, repeating the entire process. Moreover, it is saved the distinct client's weights of the rounds in a list called "save_weights", prepared for utilization to calculate the classification metrics in test and the weight divergence.

```python
def RAR_architecture(n_times, initial_weights, n_i, model, data_dict):
    """
    Ring All-Reduce architecture.

    :param n_times: Number of communication rounds
    :param weights_clients: Initial weights for each client
    :param n_i: List with number of samples for each client
    :param model: optimal ANN model
    :param data_dict: Dictionary of client data with train and test split
    :return save_weights: Aggregated model weights of all the rounds
    """
    for i in range(1,n_times+1):
        # Each round begins with weights of one model per client.
        for idx in range(len(clients)):
            # Setting initial weights of the round to each client
            model.set_weights(weights_clients[idx][0])
            X_train_np = np.array(data_dict[idx]['X_train'])
            y_train_np = np.array(data_dict[idx]['y_train'])
            # Training with optimal parameters found in the design of
            # the ANN architecture
            model.fit(X_train_np,y_train_np, epochs=38, batch_size=14)

            # The new weights are added to the list stored at
            # position 'i' of weights_clients
            weights_clients[clients[idx]-1].append(model.get_weights())

            # Add those weights to a list stored in position 'j',
            # where 'j' represents all the clients with whom
            # client 'i' communicates
            weights_clients[clients_send[idx]-1].append(model.get_weights())

        # Averaging them will provide one set of weights again for
        # each position in weights_clients.
        for idx in range(len(clients)):
            # Calculating the lengths
            len_1 = len(data_dict[clients[idx]-1]['X_train'])
            len_2 = len(data_dict[clients_send[idx]-1]['X_train'])
            samples_len = [len_1,len_1]
            # Aggregate the weights obtained with each client
            avg_weights = ave_weights(samples_len,weights_clients[idx])
            weights_clients[idx] = [avg_weights]

        # Save actual weights of the client in a round
        save_weights.append(weights_clients.copy())
    return save_weights
```

Code Block 3.4: Simulated RAR architecture algorithm in Python code.

Architectures with two-to-one communications end up having three client's weights per position in the "weights_clients" before using the aggregation function.

It is worth noting the Code Block 3.5, where it is shown how the list "clients" is randomly modified each round, with the logical condition that a client does not send weights to itself. This is the only difference between the code of the random GL architecture and the RAR architecture.

```
def Random_GL_architecture(n_times, initial_weights, n_i, model, data_dict):
    """
    Random Gossip Learning architecture.

    :param n_times: Number of communication rounds
    :param weights_clients: Initial weights for each client
    :param n_i: List with number of samples for each client
    :param model: optimal ANN model
    :param data_dict: Dictionary of client data with train and test split
    :return save_weights: Aggregated model weights of all the rounds
    """
    for i in range(1,n_times+1):
    # At the beginning of each round, I shuffle "clients" in a way
    # that a client does not send weights to itself (in "clients_send")
    while True:
        rn.shuffle(clients)
        result = list(map(lambda x,y: x-y,clients_send,clients))
        # counter += 1
        if 0 not in result:
            # print(counter)
            # print(clients)
            break

        # Each round begins with weights of one model per client.
        for idx in range(len(clients)):
            ...

        # Averaging them will provide one set of weights again for
        # each position in weights_clients.
        for idx in range(len(clients)):
            ...

        # Save actual weights of the client in a round
        save_weights.append(weights_clients.copy())
    return save_weights
```

Code Block 3.5: Simulated random GL architecture algorithm in Python code.

3.3 Hardware and software used

The versions of the *software* packages or libraries where the developed code relies on to run properly are:

- Python [39]: 3.8.10
- NumPy [40]: 1.23.4
- Pandas [41]: 2.0.3
- Matplotlib [42]: 1.23.4
- Scikit-learn [36]: 1.3.2
- TensorFlow [43]: 2.11.0

The specific modules imported are listed at the start of each notebook on the GitHub repository: `https://github.com/mma735/TFM-DS.git`. The code is licensed under the Apache 2.0 License and a *requirements.txt* file is also provided with the dependencies needed.

Due to the complexity of the problem in terms of the neural network refinement process and the optimization of the hyperparameters, as well as the numerous tests carried out to optimize the distributed architecture, it was necessary to use specific hardware. In particular, the use of a GPU was necessary to accelerate the computations. Specifically for this purpose, a deployment has been made on the AI4EOSC platform with the following *hardware* specifications:

- GPU: NVIDIA Tesla V100
- RAM: 16 GB
- Number of CPUs: 8
- Platform: AI4EOSC

AI4EOSC is a cutting-edge platform specifically designed to leverage the power of AI, DL, and ML technologies within the context of the European Open Science Cloud (EOSC). This platform streamlines the utilization of advanced AI techniques for research and innovation purposes, providing users with a comprehensive suite of tools and services to efficiently handle large distributed datasets [44].

Chapter 4

Results and discussion

This chapter shows and discuss the results of running the distributed learning architectures (Section 3.2.2) with the optimal ANN model architecture found. As mentioned at the beginning of the Section 3.2, measures of the performance of these approaches implemented in Python code, including FL, RAR, conditional GL, random GL and the customized one, was evaluated using classification metrics and the weight divergence customized equation (Equation 3.1). Also it is worth noting to remember that *validation* was performed during hyperparameters optimization when searching for an optimal ANN model, as explained in Section 3.2.1.

4.1 Classification metrics evaluation

In all the cases the maximum number of rounds was set to 100, which has been verified in trials to be sufficient for achieving convergence of the classification metrics. These metrics were complemented in client-to-client architectures with the weight convergence metric defined in the Equation 3.1.

The results for the average metrics evolution in the test over 100 rounds for the approaches are shown in Figure 4.1.

The evolution of test metrics aligns with expectations, demonstrating an improvement as the communication rounds progress. That is, with the designed approaches, the weights of each client's models efficiently communicate among themselves, enhancing the model of each client. *This consistency suggests a robust implementation of the aggregation method into the distributed architectures, validating their efficacy.*

In terms of the loss metrics, the architectures show early convergence (Figure 4.1.a), around 40 communication rounds. The accuracy quickly reaches a stable value of 0.8 (Figure 4.1.b), where FL presents in this case slightly worse performance compared to the others architectures analyzed. The same applies to the AUC metric (Figure 4.1.c), which immediately reaches a value near 0.9, and little differences can be observed between them. However, a large number of simulations (e.g. 100 simulations of 100 rounds each) should be conducted to obtain a statistical error range in these graphics and extract conclusions.

Figure 4.1 shows the complete simulation of communication rounds, but some discussion could be focused on the early rounds, as shown in Figure 4.2. Although more simulations must be conducted to validate these results, as mentioned before, it can be observed that FL demonstrates faster convergence in terms of communication rounds for the loss metric (Figure 4.2.a) compared to the other approaches, with RAR being the slowest. This observation aligns with expectations, as mentioned in Section 1.3.2, where FL involves all clients combining their weights in one communication round, while RAR only involves two clients communicating in each round.

However, the difference in convergence in the approaches is only around ten rounds (Figure 4.2.a). Then, using a distributed architecture without the presence of a central server instead of one with a server would not make much difference, in this case where there are 5 clients, in terms of the number of rounds needed for the client's model to converge, but the advantages of these kind of architectures would be obtained, as explained in Section 2.3. Even a random GL approach could potentially achieve faster convergence in terms of time, as it does not need to synchronize the updates of the client network like FL.

These hypothesis are not valid for experiments with a larger number of clients; intuition suggests that the FL approach should increase the advantage in terms of convergence velocity with respect to the other studied approaches as the number of clients increases.

For the purpose of this work, the implementation of the distinct distributed architectures is satisfactorily fulfilled, reflecting their differences in the test metrics as commented and shown in the Figure 4.2.

Additionally, some interesting values presented in early rounds in the Figure 4.2 are collected in form of tables, in the Table 4.1, the Table 4.2 and in the Table 4.3, corresponding with the Figure 4.2a, Figure 4.2b and Figure 4.2c respectively.

The evolution of the average loss function values across different architectures can be observed in Table 4.1, where the FL architecture demonstrates a more rapid decrease, closely followed by the customized architecture, which reaches a value in the tenth round only 21% higher. In comparison, values at least 60% greater relative to that of the FL architecture are observed in the tenth round for the other approaches, including RAR, conditional, and random GL approaches. This is consistent with the observation that increased communication between clients within a round leads to faster convergence of the loss function.

Similarly, as can be seen both in Table 4.2 and Figure 4.2b, results for average accuracy exhibit the same trend. Notably, the difference in values in the first round between the FL approach and the other approaches is greater than that seen in the average loss function values. For instance, in the first round, the FL approach achieves a value 40% higher than that obtained by the random GL approach. It is important to note that the FL approach achieves an accuracy of 80% in the first round, which increases only slightly, by about 5%, in subsequent rounds. However, it is crucial to conduct further simulations to obtain more statistically reliable results. This last could clarify the tendency of both numerical (Table 4.3) and graphical (Figure 4.2c) results for average AUC, as they show similar values even for the first round.

Round	*RAR*	*GL_conditional*	*GL_random*	*customized*	*FL*
1	4.37	3.89	3.93	3.89	3.75
2	4.01	3.27	3.26	3.32	3.11
3	3.62	2.8	3.02	2.82	2.7
4	3.24	2.42	2.52	2.34	2.37
5	2.95	2.11	2.4	1.97	2.0
6	2.6	1.87	2.14	1.66	1.67
7	2.22	1.67	1.85	1.4	1.31
8	1.88	1.51	1.63	1.19	1.01
9	1.59	1.37	1.67	1.06	0.84
10	1.4	1.27	1.45	0.96	0.79

Table 4.1: *Average loss values* in early rounds for the developed approaches.

Round	*RAR*	*GL_conditional*	*GL_random*	*customized*	*FL*
1	0.55	0.62	0.48	0.56	0.8
2	0.56	0.74	0.57	0.63	0.83
3	0.75	0.84	0.67	0.78	0.82
4	0.81	0.84	0.8	0.82	0.83
5	0.81	0.86	0.84	0.8	0.83
6	0.82	0.86	0.84	0.79	0.84
7	0.83	0.86	0.85	0.8	0.83
8	0.84	0.86	0.84	0.8	0.83
9	0.84	0.84	0.84	0.8	0.82
10	0.84	0.85	0.84	0.8	0.83

Table 4.2: *Average accuracy values* in early rounds for the developed approaches.

4.2 Weight divergence evaluation

The weight divergence could provide insights into how effectively the clients communicate during the rounds. At the convergence point of all approaches, which is reached around round 50, this function is calculated for each client, and the resulting values are presented in Table 4.4. It is important to note that the FL approach does not appear in this table because, according to its definition (see Equation 3.1), the value of the weight divergence for all clients remains 0 throughout the communication rounds, as they aggregate the model weights together in each communication round.

Round	*RAR*	*GL_conditional*	*GL_random*	*customized*	*FL*
1	0.85	0.87	0.88	0.87	0.89
2	0.9	0.91	0.9	0.88	0.9
3	0.92	0.91	0.91	0.88	0.9
4	0.91	0.9	0.92	0.88	0.9
5	0.91	0.9	0.9	0.89	0.9
6	0.91	0.89	0.91	0.89	0.9
7	0.91	0.9	0.91	0.9	0.9
8	0.9	0.89	0.91	0.91	0.89
9	0.9	0.89	0.91	0.89	0.9
10	0.9	0.9	0.9	0.89	0.89

Table 4.3: *Average AUC values* in early rounds for the developed approaches.

The maximum value is given by the pair of clients 2-5 in the conditional-GL approach, but this pair reduces the divergence notably in the customized and the random-GL approaches. This could be attributed to the long distance that weights have to "travel" along the communication rounds in the conditional-GL architecture, but in the other two, weights can communicate directly (see Figure 3.4). This also occurs with the pair 1-5, which has relatively low values in all the architectures because these clients are, at most, separated by one client in the conditional-GL approach.

However, the distance of communication is not the only determinant factor. The similarity of the data is also important, as explained in Section 2.3. An example of this can be seen in the values of the pair 2-3; they are directly communicated in RAR and conditional GL approaches, but the values of divergence are high (see Figure 3.4). These values show a notable reduction when they aggregate their weights with other clients beforehand, as occurs in the random GL or when other clients are included in the aggregation, obtaining a more efficient architecture, as in the customized approach.

Then, these results are in accordance with the importance of designing a distributed architecture without a central server. Performing this kind of analysis could be essential in a real-world scenario. Specifically in this work, it has been demonstrated that the architecture design has an impact on the "Heart Disease" dataset studied.

Finally, the Figure 4.3 presents the results of the calculation of the average of the weights divergence for all the communication rounds of the clients performed in the different architectures.

The evolution of the average weight divergence is as expected. Initially, clients receive the same weights from the ANN model, and each client develops a distinct trajectory of the "average weight divergence" based on the communication architecture. Values show a slight stabilization when the model converges.

Pair of clients	*RAR*	*conditional-GL*	*random-GL*	*customized*
1-2	0.79	0.81	0.44	0.27
1-3	0.56	**0.14**	0.26	0.35
1-4	0.48	0.46	0.24	0.19
1-5	**0.02**	0.34	**0.04**	**0.03**
2-3	**0.88**	0.88	0.23	0.15
2-4	**0.88**	0.76	**0.46**	0.21
2-5	0.79	**0.97**	0.43	0.27
3-4	0.51	0.49	0.25	0.23
3-5	0.56	0.20	0.25	**0.37**
4-5	0.48	0.58	0.21	0.21

Table 4.4: Weight divergence of the architectures in round 50, where all models converge in terms of the loss function.

The most notable aspect that we can note in Figure 4.3 is the significant difference in values taken by the architectures, which can be distinguished into two groups. The first group of architectures, which take high values, consists of the conditional GL and the RAR, while the second group, which take much lower values, is made up of the customized and random GL architectures. This means that in the first group, the clients' models converge to less similar solutions than in the second group.

Looking at the design of these architectures (see Figure 3.4), the result of these average values of the weights divergence can be attributed to the lack of intercommunication among the clients throughout the rounds. The conditional GL and RAR architectures only communicate one-on-one with the same clients throughout the rounds, while the customized architecture achieves better intercommunication among clients by communicating two-on-one, and the random GL achieves this by changing one-on-one communication randomly throughout the rounds. This mode of communication of the random GL also causes greater fluctuation in the values of the average weight divergence, but they are still much lower than those of the architectures in the first group.

Again, the most efficient architecture turns out to be the random GL, as it achieves better uniformity of the clients' models with simple communication (one-on-one), which, as discussed in Section 2.3, better suits a real-life scenario where the topology changes continuously, with clients entering and exiting throughout the communication rounds.

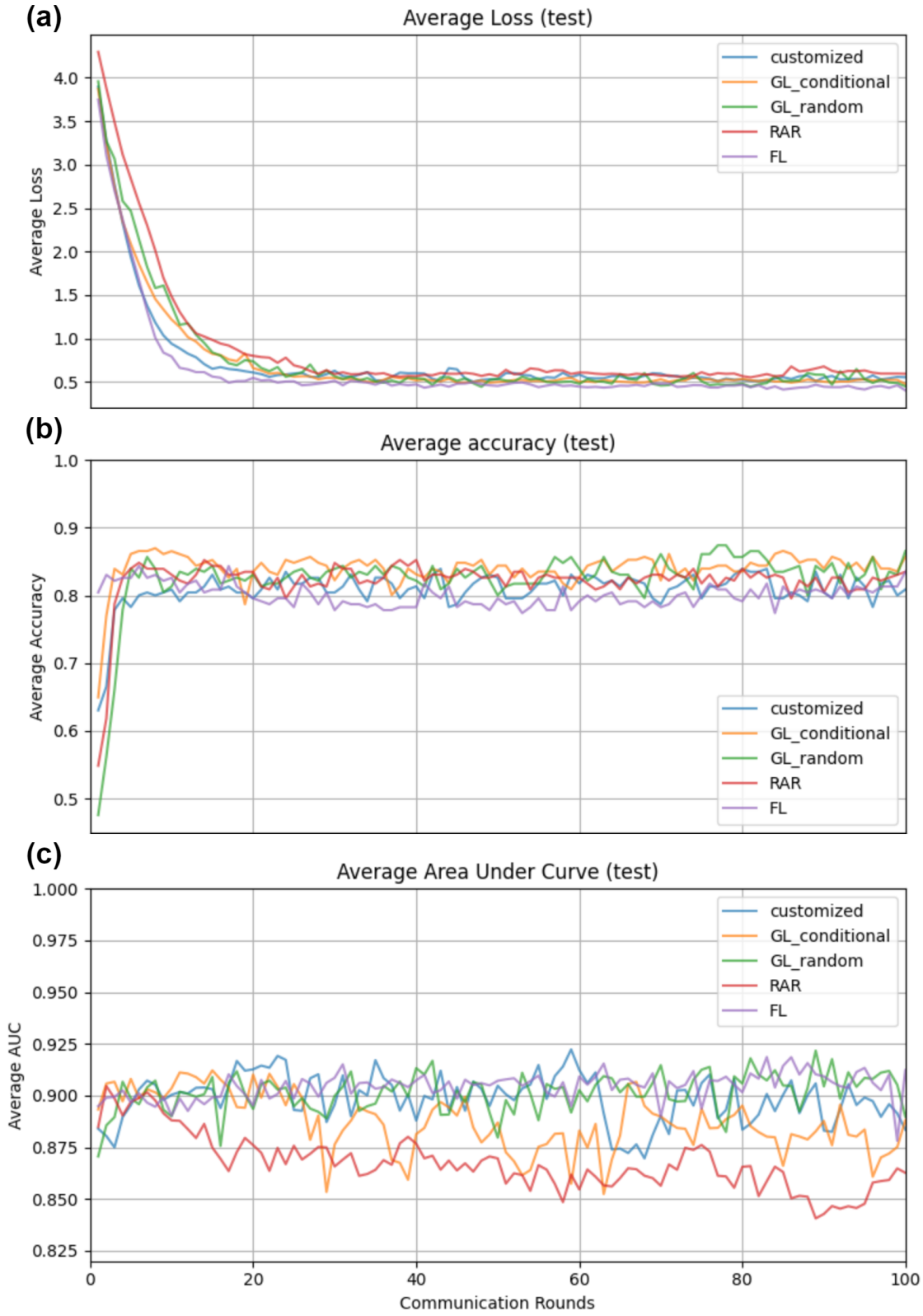


Figure 4.1: Average performance metrics, including (a) loss, (b) accuracy, and (c) area under the curve, are computed across all clients for each simulated approach over 100 rounds.

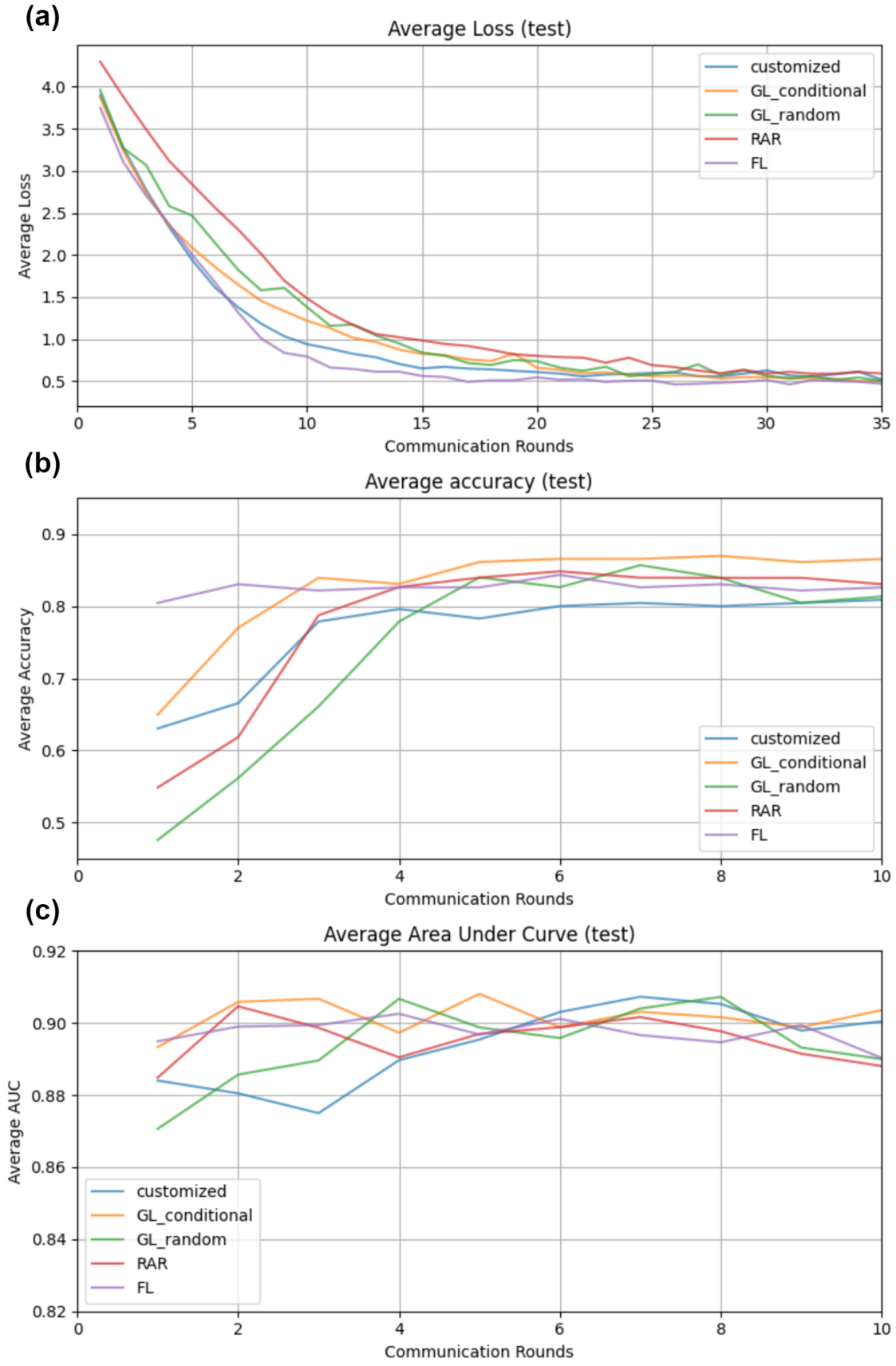


Figure 4.2: Average performance metrics, including (a) loss, (b) accuracy, and (c) area under the curve, across all clients for each approach during the initial rounds.

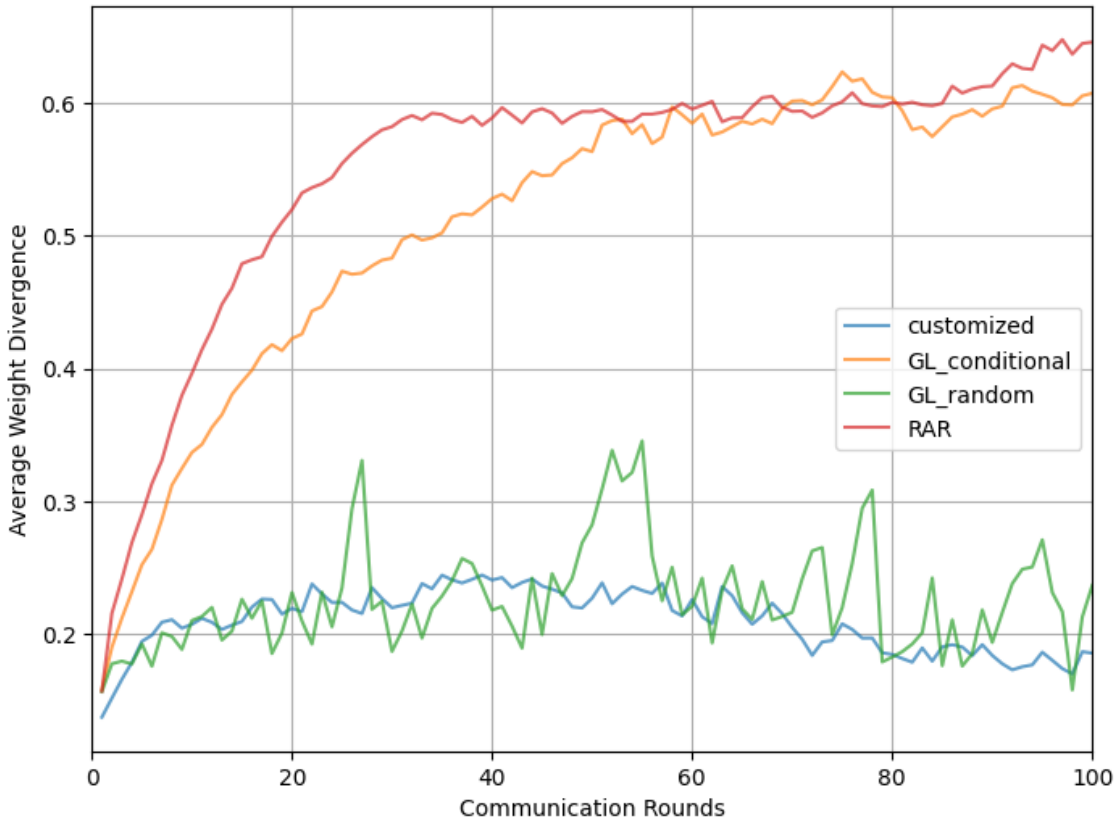


Figure 4.3: The difference of the average weight divergence between distinct distributed learning approaches over 100 communication rounds.

Chapter 5

Conclusions

Based on the objectives proposed, the conclusions of this work revolve around the comprehensive exploration and evaluation of various distributed learning techniques and the methodology followed. The study has effectively compared the performance of different distributed learning techniques, including Federated Learning, Ring All-Reduce, Gossip Learning, and a customized architecture. Through this comparative analysis, strengths and weaknesses of each technique has been identified, providing valuable insights into their applicability in real-world scenarios.

Moreover, the investigation and development of distributed learning approaches has been successfully carried out, showcasing the steps required to design such approaches from scratch. The study not only explored various methodologies but also implemented them in Python, demonstrating practical insights into the procedural steps involved in designing distributed learning systems.

Furthermore, the application of distributed learning approaches to diverse datasets, especially those containing sensitive information, has been conducted with meticulous attention to privacy concerns. By applying these techniques to openly available datasets with distinct clients, the study has showcased the practical feasibility of distributed learning techniques in scenarios involving sensitive data. Moreover, the study has reviewed within the analysis of the State of the Art existing security measures such as differential privacy, homomorphic encryption, and blockchain to reinforce data security in such scenarios.

Additionally, the search for an effective Artificial Neural Network (ANN) architecture for all clients yielded positive results in terms of model performance. The procedural steps involved in searching for an optimal ANN architecture in a real-world scenario has been described, providing valuable insights into model selection and evaluation.

Finally, the implementation of simulated distributed learning techniques in Python, along with related methods, provided a robust approach for evaluating their performance. Through rigorous testing and evaluation, the implementation contributed to understanding the communication efficiency and convergence behavior of distributed learning architectures in the specified dataset. The availability of the entire commented Python code in the GitHub repository[1] and in Zenodo[2] further facilitates replication and validation of the study's findings.

[1] `https://github.com/mma735/TFM-DS.git`
[2] `https://doi.org/10.5281/zenodo.10671543`

Regarding future perspectives, the following research lines can be proposed in order to enhance the current work:

- Conducting additional simulations of the implemented architectures to gather substantial statistical evidence. This iterative process is crucial for refining error estimates and enriching the data's analysis. By systematically repeating these simulations, it can be ensured the reliability of our findings and uncover deeper insights, leading to a more comprehensive understanding of the underlying patterns and behaviors.

- The study of other distributed machine learning approaches with this dataset such as Split Learning, All Reduce, neighbor architecture or other architectures in which a client communicates with more than one client.

- The inclusion of additional privacy preserving techniques, such as the mentioned above: differential privacy (either during local training or in the aggregation of parameters), homomorphic encryption or blockchain.

- Exploration of various AI models, particularly their adaptation to diverse machine learning models beyond neural or convolutional networks.

Bibliography

[1] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *CoRR*, abs/2003.06307, 2020. (Cited in pages 7 y 8.)

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014. (Cited in page 8.)

[3] René Mayer and Hans-Arno Jacobsen. Distributed and parallel machine learning: A comprehensive review. *Journal of Parallel and Distributed Computing*, 2020. (Cited in page 9.)

[4] Weixing Su, Linfeng Li, Fang Liu, Maowei He, and Xiaodan Liang. Ai on the edge: a comprehensive review. *Artificial Intelligence Review*, 55(8):6125–6183, 2022. (Cited in pages 9, 19, 20 y 21.)

[5] Judith Sáinz-Pardo Díaz and Álvaro López García. Study of the performance and scalability of federated learning for medical imaging with intermittent clients. *Neurocomputing*, 518:142–154, 2023. (Cited in pages 9, 11, 18, 19 y 32.)

[6] R Detrano, A Janosi, W Steinbrunn, M Pfisterer, JJ Schmid, S Sandhu, KH Guppy, S Lee, and V Froelicher. International application of a new probability algorithm for the diagnosis of coronary artery disease. *The American Journal of Cardiology*, 64(5):304–310, August 1989. (Cited in pages 9, 11 y 24.)

[7] European Commission. Horizon Europe Programme Guide. `https://ec.europa.eu/info/funding-tenders/opportunities/docs/2021-2027/horizon/guidance/programme-guide_horizon_en.pdf`, 2021. [Web; last accessed on 12-14-2024] Page: 48-49. (Cited in page 10.)

[8] ELIXIR Europe. Data life cycle, 2024. [Web; last accessed on 12-02-2024]. (Cited in pages 10 y 11.)

[9] DataONEorg. Data management lessons. `https://github.com/DataONEorg/Education/blob/master/_lessons/lessons/01_management/slides.md`, 2024. [Web; last accessed on 12-02-2024]. (Cited in page 11.)

[10] R. Detrano, A. Janosi, W. Steinbrunn, M. Pfisterer, J.J. Schmid, S. Sandhu, K.H. Guppy, S. Lee, and V. Froelicher. Heart Disease Data Set. `https://web.archive.org/web/20010622230020/http://www.ics.uci.edu/pub/machine-learning-databases/heart-disease/`, 1988. Snapshot from Internet Archive dated June 22, 2001. [Web; last accessed on 14-02-2024]. (Cited in

pages 11 y 24.)

[11] IBM. What is machine learning?, 2018. [Web; accessed on 22-01-2024]. (Cited in page 15.)

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018. (Cited in page 15.)

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. (Cited in pages 15 y 18.)

[14] Baptiste Rocca. Handling imbalanced datasets in machine learning, 2019. [Web; last accessed on 23-01-2024]. (Cited in page 16.)

[15] Simplilearn. Introduction to data imputation, 2023. [Web; last accessed on 11-02-2024]. (Cited in page 17.)

[16] Ajitesh Kumar. Missing data imputation techniques in machine learning, October 2018. [Web; last accessed on 11-02-2024]. (Cited in page 17.)

[17] Baeldung. Epoch vs. batch vs. mini-batch. `https://www.baeldung.com/cs/epoch-vs-batch-vs-mini-batch`, 2021. [Web; last accessed on 14-02-2024]. (Cited in page 17.)

[18] István Hegedűs, Gábor Danner, and Márk Jelasity. Decentralized learning works: An empirical comparison of gossip learning and federated learning. *Journal of Parallel and Distributed Computing*, 148:109–124, 2021. (Cited in pages 18, 19 y 21.)

[19] Brendan McMahan, Emily Moore, Daniel Ramage, et al. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017. (Cited in pages 19 y 20.)

[20] Sijia Wang, Toni Tuor, Theodoros Salonidis, and et al. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019. (Cited in page 19.)

[21] Amirali Reisizadeh, Aryan Mokhtari, Hamed Hassani, and et al. Fedpaq: a communication-efficient federated learning method with periodic averaging and quantization. In *International Conference on Artificial Intelligence and Statistics*, pages 2021–2031. PMLR, 2020. (Cited in page 20.)

[22] Sanaa AbdulRahman, Hala Tout, Ahmed Mourad, and et al. Fedmccs: multicriteria client selection model for optimal iot federated learning. *IEEE Internet of Things Journal*, 8(6):4723–4735, 2020. (Cited in page 20.)

[23] Yujun Lin, Song Han, Huizi Mao, and et al. Deep gradient compression: reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017. (Cited in page 20.)

[24] Sebastian Caldas, Jakub Konečny, H Brendan McMahan, and et al. Expanding the reach of federated learning by reducing client resource requirements. *arXiv preprint arXiv:1812.07210*, 2018. (Cited in page 20.)

[25] Yujing Chen, Xiaoyang Sun, and Yuanqing Jin. Communication-efficient fed-

erated deep learning with layerwise asynchronous model update and temporally weighted aggregation. *IEEE Transactions on Neural Networks and Learning Systems*, 31(10):4229–4238, 2019. (Cited in page 20.)

[26] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 603–618. ACM, 2017. (Cited in page 20.)

[27] X. Li, K. Huang, W. Yang, and et al. On the convergence of fedavg on non-iid data. *ArXiv preprint arXiv:1907.02189*, 2019. (Cited in page 20.)

[28] M. Duan, D. Liu, X. Chen, and et al. Self-balancing federated learning with global imbalanced data in mobile systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):59–71, 2020. (Cited in pages 20 y 21.)

[29] Yujie Zhao, Mingzhe Li, Liangzhen Lai, and et al. Federated learning with non-iid data. *ArXiv preprint arXiv:1806.00582*, 2018. (Cited in page 20.)

[30] N. Yoshida, T. Nishio, M. Morikura, and et al. Hybrid-fl for wireless networks: Cooperative learning mechanism using non-iid data. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020. (Cited in page 20.)

[31] Acharya Lalitha, Oguz Emre Kilinc, Tara Javidi, and et al. Peer-to-peer federated learning on graphs. 2019. (Cited in page 21.)

[32] István Hegedűs, Georg Danner, and Márk Jelasity. Gossip learning as a decentralized alternative to federated learning. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 74–90. Springer, 2019. (Cited in page 21.)

[33] C Hu, J Jiang, and Z Wang. Decentralized federated learning: a segmented gossip approach. 2019. (Cited in page 21.)

[34] Fedesoriano. Heart failure prediction dataset, September 2021. [Web; accessed on 28-04-2023]. (Cited in page 23.)

[35] Adam Rowden and Meredith Goodwin. ST Depression on ECG. *Medical News Today*, 2022. [Web; last accessed on 10-02-2024]. (Cited in page 25.)

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. (Cited in pages 25, 26, 31 y 39.)

[37] Jeff Hale. Scale, standardize, or normalize with scikit-learn, March 2019. [Web; accessed on 04-05-2023]. (Cited in page 26.)

[38] Wenyu Zhang, Xiumin Wang, Pan Zhou, Weiwei Wu, and Xinglin Zhang. Client selection for federated learning with non-iid data in mobile edge computing. *IEEE Access*, 9:24462–24474, 2021. (Cited in page 30.)

[39] Python Software Foundation. *Python: A dynamic, open source programming language*, 2021. [Web; last accessed on 14-02-2024]. (Cited in page 39.)

[40] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. (Cited in page 39.)

[41] The pandas development team. pandas-dev/pandas: Pandas, February 2020. (Cited in page 39.)

[42] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. (Cited in page 39.)

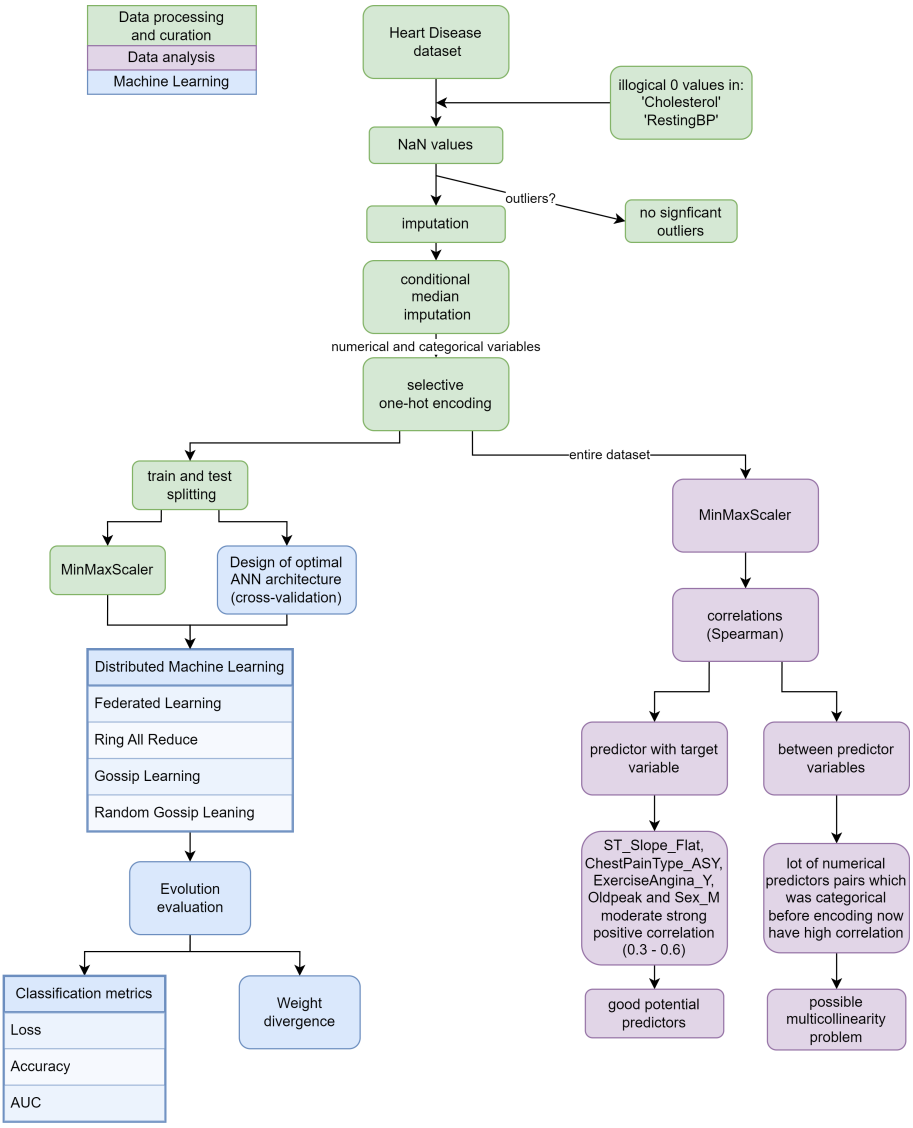
[43] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. (Cited in page 39.)

[44] Agencia Estatal Consejo Superior de Investigaciones Científicas (CSIC). AI4EOSC - Artificial Intelligence for the European Open Science Cloud. `https://ai4eosc.eu/`. [Web; last accessed on 11-02-2024]. (Cited in page 39.)

Appendix A

Diagram of the methodology implemented

The diagram illustrating the procedural steps applied to the dataset analyzed is shown in the following figure:



Appendix B

Artificial Neural Network architecture

The following diagram graphically represents the optimal ANN architecture designed.