UNIVERSITÀ DI PISA

DEPARTMENT OF COMPUTER SCIENCE

Master Degree in Computer Science

Artificial Intelligence curriculum

Federated Echo State Networks for Stress Prediction in the Automotive Use Case

Supervisors:

Prof. Davide Bacciu

Prof. Iraklis Varlamis

Dr. Valerio De Caro

Candidate:

Antonio Di Mauro

ACADEMIC YEAR 2021/2022

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

Recent years have witnessed the rapid development of the Internet of Things (IoT) which provides ubiquitous sensing and computing capabilities to connect a broad range of things to the Internet [3]. To obtain insights on the data generated from ubiquitous IoT devices, Artificial Intelligence (AI) techniques such as Deep Learning (DL) have been widely exploited to train data models for enabling intelligent IoT applications such as smart healthcare, smart transportation and smart city [4].

Traditionally, AI functions are placed in a cloud server or a data center for data learning and modeling [5], which incurs critical limitations given the huge amount of data from IoT sources. According to Cisco, there are nearly 850 ZB of data generated by all people, machines and things at the network edge by 2021. In sharp contrast, the global data center traffic is presumed reaching 20.6 ZB in 2021 [6]. With such a tremendous growth of IoT data at the network edge, in line with the growth in the number of devices (Fig. 1.1), centralizing the computation on the remote servers may be infeasible due to the required network resources and the incurred latency.

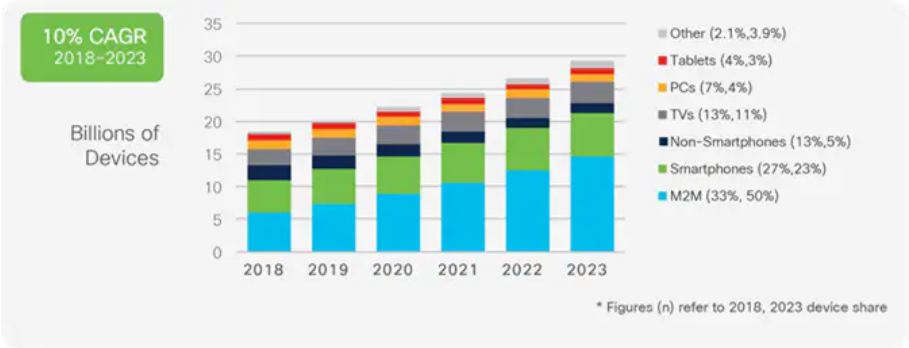

Figure 1.1: Global device and connection growth [1]

Furthermore, the use of third-party servers for AI training also raises privacy concerns, as training data may contain sensitive information such as user addresses or personal preferences [7]. It is thus highly necessary for developing innovative AI approaches to realize efficient and privacy-enhanced intelligent IoT networks and applications.

Concurrently, both industry and society are experiencing the transformational impact of the autonomous systems revolution, empowered by automation capabilities offered by Artificial Intelligence (AI). Cyber-physical Systems of Systems (CPSoS) define a multi-faceted and dynamic environment where autonomy is fundamental to govern the complexity of interactions between the virtual and physical worlds with minimal human intervention. However, even when the most advanced degree of autonomy is exercised, the human is a variable which cannot be left out of the CPSoS equation, particularly in safety critical scenarios like autonomous transportation [8].

In this context, the TEACHING project[1] (i.e., a computing Toolkit for building Efficient Autonomous appliCations leveraging Humanistic INtelliGence) is an EU-funded project that puts forward a vision of humans at the center of autonomous CPSoS. Embracing the concept of Humanistic Intelligence, it pursues a vision where the CPSoS adapts itself by means of human feedback, achieving a mutual empowerment towards a shared goal.

The objective of the Project is the design, development and deployment of autonomous, adaptive and dependable CPSoS applications, al-

[1] https://teaching-h2020.eu/

lowing them to exploit a sustainable human feedback to drive, optimize and personalize the provisioning of their services.

This Work was done under the Human State Monitoring task of the TEACHING project, and aims to develop an efficient learning model for predicting the stress conditions of a human driver from physiological data. Being such data inherently distributed and privacy-constrained, the further challenge that we addressed was to perform the learning in a federated environment.

Specifically, this Work tries to improve the State of the Art (SOTA) performances of an Echo State Network (ESN) used for the prediction of the stress value of a human driving car, which acts as a proxy for adapting the driving profile. The final model is analyzed in a FL scenario to compare different FL algorithms and comparing them with SOTA performances.

In Chapter 2 are analyzed the theoretical aspects concerning the ESN model, FL algorithms and FL applied to ESNs. In Chapter 3 is explained the specific implementation of the FL algorithm FedCurv used on ESNs. In Chapter 4 is discussed the novel approach of discriminating among anomalous/not anomalous client in the federated setting. Chapter 5 is focused on the preprocessing of the WESAD Dataset[2] used for the experiments that are explained in Chapter 6. The conclusions and future works are discussed in Chapter 7.

[2] https://archive.ics.uci.edu/ml/datasets/WESAD+%28Wearable+Stress+and+Affect+Detection%29

Chapter 2

Background

In this Chapter are analyzed theoretical aspects behind Reservoir Computing (RC), with a focus on Echo State Networks (ESNs). Next, we will address Federated Learning (FL) along with two different FL algorithms: `FedAvg` and `FedCurv`. Finally, we will discuss the specific aspect of FL applied to ESNs.

2.1 Reservoir Computing and Echo State Networks

Artificial Recurrent Neural Networks (RNNs) represent a large and varied class of computational models that are designed in analogy with biological brain modules. In an RNN numerous abstract neurons (also called units or processing elements) are interconnected by likewise abstracted synaptic connections (or links), which enable activations to propagate through the network. The characteristic feature of RNNs that distinguishes them from the feedforward neural networks is that the connection topology possesses feedback loops. Such loops enable the RNNs to may develop a self-sustained temporal activation dynamics along its recurrent connection pathways, even in the absence of input. Mathematically, RNNs enrich feedforward neural networks by introducing the temporal component in the equation, making it a dynamical system modelled by its corresponding differential equation. If driven by an input signal, an RNN preserves in its internal state a nonlinear transformation of the input history, in other words, it has a dynamical memory, and is able to process temporal context information [9].

The main shortcoming of RNNs is that they are difficult to train by gradient-descent-based methods, which aim at iteratively reducing the training error. The reasons behind such difficulty are twofold. First, gradual change of network parameters during learning drives the network dynamics through bifurcations [10]. At such points, the gradient information degenerates and may become ill-defined. As a consequence, convergence cannot be guaranteed. Second, computing the gradient of the error requires unfolding the network over time, which increases the computational cost of the learning algorithm and may make it unfeasible depending on the size of the model and the supporting hardware. Another shortcoming is represented by the difficulty of learning long-term dependencies, since the necessary gradient information exponentially dissolves over time [11]. Long Short-Term Memory (LSTM) networks represents a possible solution [12]. However, besides the difficulty of handling its corresponding learning algorithms [9], this model still falls back in the shortcoming described above.

In this situation of slow and difficult progress, in 2001 a fundamentally new approach to RNN design and training was proposed by Wolfgang Maass under the name of Liquid State Machines [13] and by Herbert Jaeger under the name of Echo State Networks (ESNs) [14]. This approach is often referred to as Reservoir Computing (RC). The RC paradigm avoids the shortcomings of gradient-descent RNN training stated above, by setting up RNNs in the following way:

- A Recurrent Neural Network is *randomly* created and remains unchanged during training. This RNN is called the *reservoir*. It is passively excited by the input signal and maintains in its state a nonlinear transformation of the input history.

- The desired output signal is generated as a linear combination of the neuron's signals from the input-excited reservoir. This linear combination, named *readout*, is obtained by linear regression, using the teacher signal as a target.

2.1.1 Echo State Network Formalization

Let a *problem* or a *task* in our context of machine learning be defined as a problem of learning a functional relation between a given input $\mathbf{u}(t) \in \mathbb{R}^{N_u}$ and a desired output $\mathbf{y}_{target}(t) \in \mathbb{R}^{N_y}$, where $t = 1, \ldots, T$, and T is the number of data points in the training *dataset* $\{(\mathbf{u}(t), \mathbf{y}_{target}(t))\}$. In a *temporal task* the function to be learned depends on the history of the input, so the expansion function has memory: $\mathbf{x}(t) = x(\ldots, \mathbf{u}(t-1), \mathbf{u}(t))$, i.e., it is an expansion of the current input and its (potentially infinite) history. Since this function has an unbounded number of parameters, practical implementations often take an alternative, recursive, definition:

$$\mathbf{x}(t) = x(\mathbf{x}(t-1), \mathbf{u}(t)). \tag{2.1}$$

The type of Recurrent Neural Networks that we will consider in this Work is a straightforward implementation of Eq. 2.1 where a nonlinear expansion with memory here leads to a *state vector* of the form:

$$\mathbf{x}(t) = f(\mathbf{W}_{in}\mathbf{u}(t) + \widehat{\mathbf{W}}\mathbf{x}(t-1)), t = 1, \ldots, T. \tag{2.2}$$

where $\mathbf{x}(t) \in \mathbb{R}^{N_x}$ is a vector of reservoir neuron activations at a time step t, $f(\cdot)$ is the neuron activation function, usually the $tanh(\cdot)$, $\mathbf{W}_{in} \in \mathbb{R}^{N_x \times N_u}$ is the input weight matrix, and $\widehat{\mathbf{W}} \in \mathbb{R}^{N_x \times N_x}$ is a weight matrix of internal network connections. The network is usually started with the initial state $\mathbf{x}(0) = 0$. The *readout* of the network is implemented in this way:

$$\mathbf{y}(t) = f_{out}(\mathbf{W}\mathbf{x}(\mathbf{u}(t))) \tag{2.3}$$

where $f_{out}(\cdot)$ is a nonlinear function (e.g., a sigmoid applied elementwise), $\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$ are the trained output weights.

Echo State Networks [15] represent one of the two pioneering reservoir computing methods. The approach is based on the observation that if a random RNN possesses certain algebraic properties, training only a linear readout from it is often sufficient to achieve excellent performance in practical applications. The untrained layer of the ESN is called a *dynamic reservoir*, and the resulting states $\mathbf{x}(t)$ are termed *echoes* of its

input history [14], this is where Reservoir Computing draws its name from.

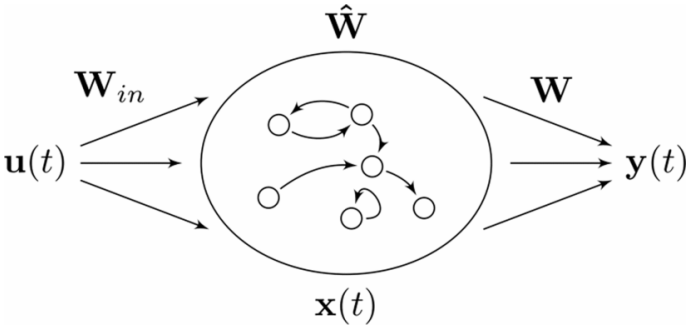

Figure 2.1: Architecture of an ESN. The input signal $\mathbf{u}(t)$ is fed to the recurrent reservoir. Then, a state $\mathbf{x}(t)$ is extracted from the reservoir, from which an output $\mathbf{y}(t)$ is computed [2]

A fundamental constraint of the ESNs is that we must ensure the stability of the dynamical system. Such constraint is satisfied by the *Echo State Property* (ESP) [14] (Def. 2.1.2). Apply the sufficient condition (Theorem 2.1.1) for the ESP is fine in theory, but often impractical because it is too strong. Usually, the necessary condition (Theorem 2.1.2) is used as an easy way for initialization of the reservoir.

The necessary condition states that the effect of a previous state $\mathbf{x}(t)$ and a previous input $\mathbf{u}(t)$ on a future state $\mathbf{u}(t+k)$ should vanish gradually as time passes (i.e. $k \to \infty$), and not persist or even get amplified [9]. For most practical purposes, the ESP is assured if the reservoir weight matrix $\widehat{\mathbf{W}}$ is scaled so that its spectral radius $\rho(\widehat{\mathbf{W}})$ (i.e., the largest absolute eigenvalue) satisfies $\rho(\widehat{\mathbf{W}}) < 1$ [14] or, using another term, $\widehat{\mathbf{W}}$ is *contractive*. Practically, the initialization of $\widehat{W}$ is made in the following way:

$$\widehat{\mathbf{W}} \leftarrow \widehat{\mathbf{W}} \frac{\rho_{desired}}{\rho(\widehat{\mathbf{W}})} \tag{2.4}$$

where $\rho_{desired} < 1$. Also the $\mathbf{W}_{in}$ matrix could be initialized in the following way:

$$\mathbf{W}_{in} \leftarrow \omega_{in} \mathbf{W}_{in} \tag{2.5}$$

where ω_{in} is the *input scaling* parameter that makes $\mathbf{W}_{in}$ values falling in the interval $[-\omega_{in}, \omega_{in}]$ if the elements of $\mathbf{W}_{in}$ are generated randomly from a uniform distribution on the interval $[-1, 1]$. The spectral radius $\rho(\widehat{\mathbf{W}})$ and the input scaling ω_{in} are key hyper-parameters of the reservoir initialization.

Definition 2.1.1 (Reservoir State Transition Function)**.** *Given N_u the input dimension and N_x the number of reservoir neurons. The reservoir is a discrete-time input-driven dynamical system, and its dynamics are driven by the State Transition Function:*

$$F : \mathbb{R}^{N_u} \times \mathbb{R}^{N_x} \to \mathbb{R}^{N_x}$$

$$\boldsymbol{x}(t) = F(\boldsymbol{u}(t), \boldsymbol{x}(t-1)) \tag{2.6}$$

where F takes an input and a state and returns a new state. The final state after seeing an entire input sequence s, given the initial state, is given by the iterative version of the State Transition Function:

$$\widehat{F} : (\mathbb{R}^{N_u})^* \times \mathbb{R}^{N_x} \to \mathbb{R}^{N_x}$$

$$\widehat{F}(s, x_0) = \begin{cases} x_0 & \text{if s=[ ]} \\ F(x(t), \widehat{F}([x(1), \ldots, x(t-1)], x_0)) & \text{if s=[x(1),. . . , x(t)]} \end{cases} \tag{2.7}$$

Definition 2.1.2 (Echo State Property (ESP))**.** *Given N_u the input dimension, N_x the number of reservoir neurons and the State Transition Function $\widehat{F}$ (Def. 2.1.1). An ESN satisfies the ESP whenever:*

$$\forall s \in (\mathbb{R}^{N_u})^N, \forall x_0, z_0 \in \mathbb{R}^{N_x} :$$

$$\left\| \widehat{F}(s, x_0) - \widehat{F}(s, z_0) \right\| \to 0, \quad as \quad N \to \infty \tag{2.8}$$

briefly, given the input sequence and two initial states, the distance between the final stages goes to 0 with the length of the input.

Theorem 2.1.1 (Sufficient Condition for the ESP)**.** *If the maximum singular value of $\widehat{W}$ is less than 1 the ESN satisfies the ESP for any possible input.*

$$\sigma_{max}(\widehat{W}) = \left\|\widehat{W}\right\|_2 < 1 \tag{2.9}$$

in other therms, there is a contractive dynamics for every input.

Theorem 2.1.2 (Necessary Condition for the ESP)**.** *If the spectral radius of $\widehat{W}$ is not smaller than 1 the ESN does not satisfies the ESP.*

$$\rho(\widehat{W}) = \max(abs(eig(\widehat{W})) < 1 \tag{2.10}$$

in other therms, there is a globally asymptotically stable dynamics around the 0 state.

ESNs (Def. 2.1.3) standardly computes the reservoir states as in Eq. 2.2, where the nonlinear function $f(\cdot)$ is a sigmoid, usually the $tanh(\cdot)$ function. Leaky integrator neuron models represent another frequent option for ESNs (Def. 2.1.4), and is the one used in this Work and named Li-ESNs.

Definition 2.1.3 (ESN)**.** *Given N_u the input dimension, N_x the number of reservoir neurons, N_y the output dimension and $\boldsymbol{x}(0) = 0$, the ESN at each time step $t \in \{1, \ldots, T\}$ computes the state vector $\boldsymbol{x}(t) \in \mathbb{R}^{N_x}$ for a given input signal $\boldsymbol{u}(t) \in \mathbb{R}^{N_u}$ as:*

$$\boldsymbol{x}(t) = \tanh(\boldsymbol{W}_{in}\boldsymbol{u}(t) + \widehat{\boldsymbol{W}}\boldsymbol{x}(t-1)).$$

where $\boldsymbol{W}_{in} \in \mathbb{R}^{N_x \times N_u}$ and $\widehat{\boldsymbol{W}} \in \mathbb{R}^{N_x \times N_x}$ are fixed and untrained. The readout of the network $\boldsymbol{y}(t) \in \mathbb{R}^{N_y}$ is computed as:

$$\boldsymbol{y}(t) = sigmoid(\boldsymbol{W}x(\boldsymbol{u}(t)))$$

where $\boldsymbol{W} \in \mathbb{R}^{N_y \times N_x}$ is trainable.

Definition 2.1.4 (Li-ESN)**.** *Given N_u the input dimension, N_x the number of reservoir neurons, N_y the output dimension, $\boldsymbol{x}(0) = 0$ and $\alpha \in \mathbb{R}$ the leaking rate $(0 < \alpha \leq 1)$, the Li-ESN at each time step $t \in \{1, \ldots, T\}$ computes the state vector $\boldsymbol{x}(t) \in \mathbb{R}^{N_x}$ for a given input signal $\boldsymbol{u}(t) \in \mathbb{R}^{N_u}$ as:*

$$\boldsymbol{x}(t) = (1 - \alpha)\boldsymbol{x}(t-1) + \alpha \tanh(\boldsymbol{W}_{in}\boldsymbol{u}(t) + \widehat{\boldsymbol{W}}\boldsymbol{x}(t-1)).$$

where $\boldsymbol{W}_{in} \in \mathbb{R}^{N_x \times N_u}$ and $\widehat{\boldsymbol{W}} \in \mathbb{R}^{N_x \times N_x}$ are fixed and untrained. The readout of the network $\boldsymbol{y}(t) \in \mathbb{R}^{N_y}$ is computed as:

$$\boldsymbol{y}(t) = sigmoid(\boldsymbol{W}x(\boldsymbol{u}(t)))$$

where $\boldsymbol{W} \in \mathbb{R}^{N_y \times N_x}$ is trainable.

2.2 Federated Learning

Nowadays phones and tablets are the primary computing devices for many people [16][17]. The powerful sensors on these devices (including cameras, microphones, and GPS), combined with the fact they are frequently carried, means they have access to an unprecedented amount of data, much of it private in nature. Models learned on such data hold the promise of greatly improving usability by powering more intelligent applications, but the sensitive nature of the data means there are risks and responsibilities to storing it in a centralized location.

Recently, the concept of Federated Learning (FL) has been proposed for building intelligent and privacy-enhanced IoT systems. Technically, FL is a distributed collaborative AI approach that allows for data training by coordinating multiple devices with a central server without sharing actual datasets [18].

For instance, multiple IoT devices can act as workers to communicate with an aggregator (e.g. a server) for performing neural network training in intelligent IoT networks. More specifically, the aggregator first initiates a global model with learning parameters. Each worker downloads the current model from the aggregator, computes its model update (e.g. Stochastic Gradient Descent) by using its local dataset, and sends

the computed local update back to the aggregator.

Then, the aggregator combines all local model updates and constructs a new improved global model. By using the computing power of distributed workers, the aggregator can enhance the training quality while minimizing user privacy leakage.

Finally, the local workers download the global update from the aggregator, and compute their next local update until the global training is complete [19].

2.2.1 Federated Averaging

Google researchers, in 2017, proposed *FederatedAveraging* (or `FedAvg`) [20] algorithm that makes as contributions 1) the identification of the problem of training on decentralized data from mobile devices while complying to privacy constraints as an important research direction; 2) the selection of a straightforward and practical algorithm that can be applied to this setting; and 3) an extensive empirical evaluation of the proposed approach. `FedAvg` is a general implementation of `FedSGD` and more concretely, it combines local stochastic gradient descent (SGD) on each client with a server that performs model averaging. It is robust to unbalanced and non-IID data distributions, and can reduce the rounds of communication needed to train a deep network on decentralized data by orders of magnitude.

It addresses the key issues of client availability and unbalanced and non-IID data. It assumes also a synchronous update scheme that proceeds in rounds of communication. There is a fixed set of K clients, each with a fixed local dataset. At the beginning of each round, a random fraction C of clients is selected, and the server sends the current global algorithm state to each of these clients (e.g., the current model parameters). Each selected client then performs local computation based on the global state and its local dataset, and sends an update to the server. The server then applies these updates to its global state, and the process repeats.

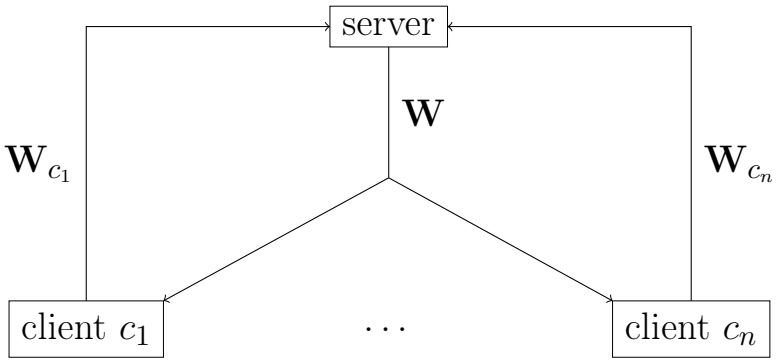

Figure 2.2: Federated Averaging Scheme. Each client c sends their local matrix $\mathbf{W}_c$ to the server. After the aggregation of the models is performed in the form of a weighted average, the server sends back the same matrix $\mathbf{W}$ to all clients [2].

The algorithm considered is applicable to any finite-sum objective of the form:

$$\min_{w \in \mathbb{R}^d} f(w) \quad \text{where} \quad f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w). \tag{2.11}$$

For a machine learning problem, we typically take $f_i(w) = \ell(x_i, y_i; w)$, that is, the loss of the prediction on example (x_i, y_i) made with model parameters w. We assume there are K clients over which the data is partitioned, with $\mathcal{P}_k$ the set of indexes of data points on client k, with $n_k = |\mathcal{P}_k|$. Thus, we can rewrite the objective in Eq. 2.11 as:

$$f(w) = \sum_{i=1}^{K} \frac{n_k}{n} F_k(w) \quad \text{where} \quad F_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(w). \tag{2.12}$$

If the partition $\mathcal{P}_k$ was formed by distributing the training examples over the clients uniformly at random, then we would have $\mathbb{E}_{\mathcal{P}_k}[F_k(w)] = f(w)$, where the expectation is over the set of examples assigned to a fixed client k. This is the IID assumption typically made by distributed optimization algorithms; `FedAvg` refers to the case where this does not hold (that is, F_k could be an arbitrarily bad approximation to f) as the non-IID setting.

The multitude of successful applications of deep learning have almost exclusively relied on variants of SGD for optimization, and can be applied naively to the federated optimization problem. This is done selecting a C-fraction of clients on each round, and compute the gradient of the loss over all the data held by these clients. Thus, C controls the *global* batch size, with $C = 1$ corresponding to full-batch (non-stochastic) gradient descent. This is the baseline algorithm named *FederatedSGD* (or `FedSGD`).

A typical implementation of `FedSGD` with $C = 1$ and a fixed learning rate η has each client k compute $g_k = \nabla F_k(w_t)$, the average gradient on its local data at the current model w_t, and the central server aggregates these gradients and applies the update $w_{t+1} \leftarrow w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k$, since $\sum_{k=1}^{K} \frac{n_k}{n} g_k = \nabla f(w_t)$. An equivalent update is given by $\forall k, w_{t+1}^k \leftarrow w_t - \eta g_k$ and then $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$. That is, each client locally takes one step of gradient descent on the current model using its local data, and the server then takes a weighted average of the resulting models.

Once the algorithm is written this way, we can add more computation to each client by iterating the local update $w^k \leftarrow w^k - \eta F_k(w^k)$ multiple times before the averaging step. This approach is named *FederatedAveraging* (or `FedAvg`). The amount of computation is controlled by three key parameters: C, the fraction of clients that perform computation on each round; E, then number of training passes each client makes over its local dataset on each round; and B, the local minibatch size used for the client updates. With $B = \infty$ it means that the full local dataset is treated as a single minibatch. Thus, at one endpoint of this algorithm family, we can take $B = \infty$ and $E = 1$ which corresponds exactly to `FedSGD`. For a client with n_k local examples, the number of local updates per round is given by $u_k = E\frac{n_k}{B}$. The pseudo-code is given in Alg. 1.

Algorithm 1 `FedAvg`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

ServerUpdate
  $w_0 \leftarrow$ initialize()
  for `each round` $t = 1, 2, \ldots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for `each client` $k \in S_t$ **do**
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w)
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for i `from 1 to` E **do**
    for batch $b \in \mathcal{B}$ **do**
      $w \leftarrow w - \eta \nabla \ell(w; b)$
  return w

2.2.2 Federated Curvature

Federated Learning on non-IID distributions has trouble achieving good results in terms of accuracy and the number of communication rounds [20]. There is a deep parallel between the FL problem and another fundamental machine learning problem called *Lifelong Learning*. In Lifelong Learning, the challenge is to learn task A, and continue on to learn task B using the same model, but without "forgetting", without severely hurting the performance on task A. In general, it is a problem of learning tasks $A_1, A_2, \ldots$ in sequence without forgetting previously-learnt tasks for which samples are not presented anymore. In the Federated Learning context, the previous problem is translated in how to learn a task without disturbing different ones learnt on the same model. An approach to this kind of problem that preserves privacy and communication offload, may be found in *Elastic Weight Consolidation* (EWC) [21].

EWC aims to prevent catastrophic forgetting when moving from learning task A to learning task B. The idea is to identify the coordinates in

the network parameters θ that are the most informative for task A, and then, while task B is being learned, penalize the learner for changing these parameters. The basic assumption is that deep neural networks are over-parameterized enough, so that there are good chances of finding an optimal solution θ^*_B to task B in the neighborhood of previously learned θ^*_A. In order to control the stiffness of θ per coordinate while learning task B, EWC uses the diagonal of the Fisher information matrix $\mathcal{I}^*_A = \mathcal{I}_A(\theta^*_A)$ to selectively penalize parts of the parameters vector θ that are getting too far from θ^*_A [21]. This is done using the following objective:

$$\tilde{L}(\theta) = L_B(\theta) + \lambda(\theta - \theta^*_A)^T diag(\mathcal{I}^*_A)(\theta - \theta^*_A) \tag{2.13}$$

The formal justification provided for Eq. 2.13 is Bayesian: Let D_A and D_B be independent datasets used for tasks A and B. We have that:

$$\log p(\theta|D_A, D_B) = \log p(D_B|\theta) + \log p(\theta|D_A) - \log p(D_B) \tag{2.14}$$

where $\log p(D_B|\theta)$ is just the standard likelihood maximized in the optimization of $L_B(\theta)$, and the posterior $\log p(\theta|D_A)$ is approximated with Laplace's method as a Gaussian distribution with expectation θ^*_A and covariance $diag(\mathcal{I}^*_A)$.

It is also well known that under some regularity conditions, the information matrix approximates the Hessian H_L of $L(\theta)$, at $\theta = \theta^*$ [22]. By this we get a non Bayesian interpretation of Eq. 2.13:

$$\tilde{L}(\theta) \approx L_B(\theta) + \frac{1}{2}(\theta - \theta^*_A)^T H_{L_A}(\theta - \theta^*_A) \approx L_B(\theta) + L_A(\theta) \tag{2.15}$$

where $L(\theta) = L_B(\theta) + L_A(\theta)$ is exactly the loss we want to minimize.

Federated Curvature (or `FedCurv`) is an adaptation of the EWC algorithm to the Federated Learning scenario. We mark by $S = \{1, \ldots, N\}$ the N nodes as a task with local datasets $\{A_1, \ldots, A_N\}$. At this step we consider each nodes in S, instead a subset of them as in `FedAvg` (but can be extended to select a subset). At round t each node $s \in S$ optimizes the following loss:

$$\tilde{L}_{t,s} = L_s(\theta) + \lambda \sum_{j \in S \setminus s} (\theta - \tilde{\theta}_{t-1,j})^T diag(\tilde{\mathcal{I}}_{t-1,j})(\theta - \tilde{\theta}_{t-1,j}). \quad (2.16)$$

On each round t, starting from initial point $\tilde{\theta}_t = \frac{1}{N}\sum_{i=1}^{N} \tilde{\theta}_{t-1,i}$, the nodes optimize their local loss by running SGD for E local epochs. At the end of each round t, each node j sends to the rest of the nodes the SGD result $\tilde{\theta}_{t,j}$ and $diag(\tilde{\mathcal{I}}_{t,j})$ that will be used for the loss of round $t+1$.

`FedCurv` might look cumbersome and expensive to store and transmit. However by careful implementation we can avoid it, so Eq. 2.16 can be arranged as in Eq. 2.17

$$\tilde{L}_{t,s} = L_s(\theta) + \lambda\theta^T \left[\sum_{j \in S \setminus s} diag(\tilde{\mathcal{I}}_{t-1,j}) \right] \theta - 2\lambda\theta^T \sum_{j \in S \setminus s} diag(\tilde{\mathcal{I}}_{t-1,j})\tilde{\theta}_{t-1,j} + \text{const} \quad (2.17)$$

The central point needs only two additional elements to maintain and transmit, of the same size of θ, and are u_t and v_t, as shown in Eq. 2.18

$$u_t = \sum_{j \in S} diag(\tilde{\mathcal{I}}_{t-1,j}) \quad \text{and} \quad v_t = \sum_{j \in S} diag(\tilde{\mathcal{I}}_{t-1,j})\tilde{\theta}_{t-1,j} \quad (2.18)$$

The device can then construct the data needed for the evaluation of $\tilde{L}$ from u_t and v_t by subtraction.

The specific implementation of the algorithm, is presented in Chapter 3.

2.3 Federated Echo State Networks

There is a vast amount of literature regarding federated RNNs, but here we focus on the paradigm of Reservoir Computing, which allows to produce highly resource-efficient RNNs [2]. Specifically, it is used the Li-ESN as described in the Def. 2.1.4 in a FL setting. In this context we consider only the trainable weight matrix $\mathbf{W}$ and the training proceeds as follows:

1. the input sequences from the training dataset are fed to the reservoir

2. the relevant states on which the network must learn to perform predictions are collected column-wise into a matrix $\mathbf{S} \in \mathbb{R}^{N_x \times N_{train}}$, where N_{train} is the number of such states, and the associated targets are collected into the matrix $\mathbf{Y} \in \mathbb{R}^{N_y \times N_{train}}$

3. the matrix $\mathbf{W}$ is obtained as the solution to a least squares minimization problem between $\mathbf{WS}$ and $\mathbf{Y}$.

In particular, a common algorithm for a regularized solution to the least squares problem is ridge regression. In this case, if $\beta \in \mathbb{R}^+$ is the L2 regularization factor chosen by model selection, the readout weights are computed in closed form as follows:

$$\mathbf{W} = \mathbf{Y}\mathbf{S}^T(\mathbf{S}\mathbf{S}^T + \beta\mathbf{I})^{-1} \tag{2.19}$$

The method proposed as *Incremental Federated Learning* (or `IncFed`) [2], uses peculiar characteristics of ESN training allowing an optimal form of federated learning. Locally, instead of computing the readout weights, each client c computes the matrices $\mathbf{A}_c \in \mathbb{R}^{N_y \times N_x}$ and $\mathbf{B}_c \in \mathbb{R}^{N_x \times N_x}$ as follow:

$$\mathbf{A}_c = \mathbf{Y}_c\mathbf{S}_c^T \quad \text{and} \quad \mathbf{B}_c = \mathbf{S}_c\mathbf{S}_c^T + \beta_c\mathbf{I} \tag{2.20}$$

The matrices $\mathbf{A}_c$ and $\mathbf{B}_c$ are sent to the server where they get summed as in the following equations:

$$\mathbf{A} = \sum_{c \in \mathcal{C}} \mathbf{A}_c \quad \text{and} \quad \mathbf{B} = \sum_{c \in \mathcal{C}} \mathbf{B}_c \tag{2.21}$$

After the summed matrices are computed, the server can compute the optimal readout weights $\mathbf{W}$ in closed-form as follows:

$$\mathbf{W} = \mathbf{A}\mathbf{B}^{-1} \tag{2.22}$$

Notice how Eq. 2.22 is mathematically equivalent to Eq. 2.19 if all data was locally available to the server.

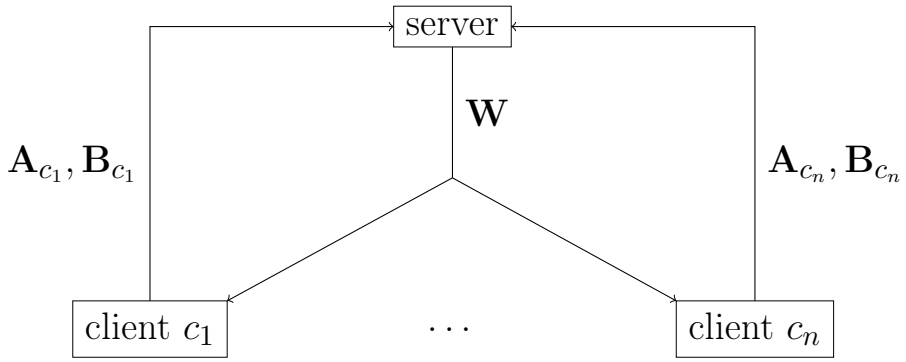

Figure 2.3: Incremental Federated Learning Scheme. Each client sends their local matrices $\mathbf{A}_c$ and $\mathbf{B}_c$ to the server. After the matrices are aggregated and multiplied to compute the optimal readout weights, the server transmits $\mathbf{W}$ back to all clients [2].

This method shows how the use of ESNs, which are efficient state-of-the-art RNN models for time-series processing, enables a form of federation that is optimal in the sense that it produces models mathematically equivalent to the corresponding centralized model. This is the baseline, or a starting point, for the FL part of this Work that tries to achieve, or even surpass, the state-of-the-art results.

Chapter 3

Federated Curvature for Echo State Networks

In this Chapter are described the federated settings and the implementation details of the Federated Curvature (`FedCurv`) algorithm applied to the ESN, following the theoretical aspects explained in Section 2.2.2.

The Federated Learning scenario of the ESNs starts initializing the server ESN model parameters $\mathbf{W}_{in} \in \mathbb{R}^{N_x \times N_u}$, $\widehat{\mathbf{W}} \in \mathbb{R}^{N_x \times N_x}$, $\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$, $u_{server} \in \mathbb{R}^{N_y \times N_x}$ and $v_{server} \in \mathbb{R}^{N_y \times N_x}$; N_x is the number of reservoir neurons, N_u is the input dimension, N_y is the output dimension, u_{server} and v_{server} are, respectively, the importance of the $\mathbf{W}$ parameters for the task at time t and the importance of the same parameters at time $t-1$. We focus our attention on $\mathbf{W}$ because they are the only parameters trained in the learning process.

The clients ESN models have the same structure of the server, the clients models parameters could be initialized as the same of the server parameters or differently from the server.

After the initialization of the federated nodes, the server broadcasts to the clients the $\mathbf{W}$, u_{server} and v_{server} parameters; u_{server} and v_{server} are initialized as zero tensors at the beginning. For each batch of local data and for each round, each client computes predictions ($\mathbf{y}(t)$) on that data. The outputs are used to penalize the local client parameters that goes too far from the previous task, and so from $\mathbf{W}$ received

from the server. To do so, we compute the gradient of the log-likelihood of the predictions with respect to the $\mathbf{W}$ parameters. The gradient is computed as $\frac{\partial log(softmax(\mathbf{y}(t)))}{\partial w}$ and it is squared and normalized by the length of the batch of data in order to get our approximation of the diagonal of the Fisher matrix; This tells us the importance of each parameter.

After computing the Fisher diagonal ($\mathcal{D}$), we need to penalize the client for going "far from the server". To do so each client considers $u = \mathcal{D}$ and $v = \mathcal{D} \times \mathbf{W}$, but in order to penalize with respect to the importance of the other clients, we need to subtract u and v, respectively, from u_{server} and v_{server}; so $u_{client} = u_{server} - u$ and $v_{client} = v_{server} - v$. The penalty term is computed as $\lambda = sum(\mathbf{W}^2 \times u_{local} - 2 \times \mathbf{W} \times v_{local})$ (as shown in Eq. 2.16) and added to the loss function to perform the optimization step and compute the $\mathbf{W}_{client}$ parameters.

After updating the client parameters, each client broadcast to the server it's own $\mathbf{W}_{client}$, u_{client} and v_{client}. The server assign to $\mathbf{W}$ the average of the $\mathbf{W}_{client}$ received from the clients and it updates u_{server} and v_{server}, respectively, summing the u_{client} and v_{client} received from the clients.

The pseudo-code of the `FedCurv` algorithm is shown in Alg. 2.

Algorithm 2 `FedCurv`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate. Each client has the reservoir parameter $\mathbf{W}_{in}$ and $\widehat{\mathbf{W}}$, α is the leaky rate, N_y is the output dimension and N_x is the number of reservoir neurons.

ServerUpdate
  let $w_0, u_0, v_0 \in \mathbb{R}^{N_y \times N_x}$
  for `each round` $t = 1, 2, \dots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for `each client` $k \in S_t$ **do**
      $w_{t+1}^k, u_{t+1}^k, v_{t+1}^k \leftarrow \text{ClientUpdate}(k,\, w_t,\, u_t,\, v_t)$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$
    $u_{t+1} \leftarrow \sum_{k=1}^{K} u_{t+1}^k$
    $v_{t+1} \leftarrow \sum_{k=1}^{K} v_{t+1}^k$

ClientUpdate$(k,\, w,\, u,\, v)$
  let $u_{local}, v_{local} \in \mathbb{R}^{N_y \times N_x}$
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for i `from 1 to` E **do**
    for batch $b \in \mathcal{B}$ **do**
      $\mathbf{x}(t) = (1-\alpha)\mathbf{x}(t-1) + \alpha \tanh(\mathbf{W}_{in} b + \widehat{\mathbf{W}}\mathbf{x}(t-1))$
      $\mathbf{y}(t) \leftarrow sigmoid(w\mathbf{x}(t))$
      $\nabla\mathcal{LL}_b \leftarrow \frac{\partial log(softmax(\mathbf{y}(t)))}{\partial w}$ ▷ Log-likelihood gradient
      $\mathcal{D}_b \leftarrow \frac{(\nabla\mathcal{LL}_b)^2}{length(b)}$ ▷ Diagonal of the Fisher matrix
      $u_b \leftarrow \mathcal{D}_b$
      $v_b \leftarrow \mathcal{D}_b \times w$
      $u_{local} \leftarrow u - u_b$
      $v_{local} \leftarrow v - v_b$
      $\lambda \leftarrow sum(w^2 \times u_{local} - 2 \times w \times v_{local})$ ▷ Penalty term
      $w \leftarrow w - \eta\nabla[\ell(w; b) + \lambda]$
  return w, u_{local}, v_{local}

The diagram of the federated process with the elements exchanged by clients and server is shown in Fig. 3.1.

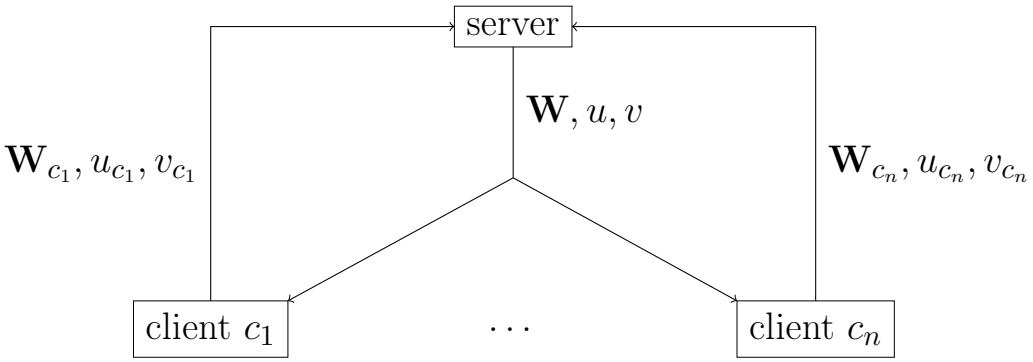

Figure 3.1: Federated Curvature Scheme. Each client c sends their local matrix $\mathbf{W}_c$, u_c and v_c to the server. After the aggregation of the models is performed in the form of a weighted average of the $\mathbf{W}_c$, the sum of the u_c and the sum of the v_c, the server sends back the same parameters $\mathbf{W}$, u and v to all clients.

The implementation has be done using Tensorflow-Federated (TFF) framework[1] due to the natural compatibility with Tensorflow (TF) framework[2] that it is used for the models implementations. TFF is an open-source framework for machine learning and other computations on decentralized data and has been developed to facilitate open research and experimentation with FL. A research FL simulation implemented in TFF typically consists of three main types of logic:

1. Individual pieces of TF code, typically `tf.function`[3], that encapsulate logic that runs in a single location (e.g., on clients or on a server). This code is typically written and tested without any `tff.*` references, and can be re-used outside of TFF

2. TFF orchestration logic, which binds together the individual `tf.function` from 1) by wrapping them as `tff.tf_computation`[4] and then orchestrating them using abstractions like `tff.federated_broadcast`[5]

[1] https://www.tensorflow.org/federated
[2] https://www.tensorflow.org/
[3] https://www.tensorflow.org/api_docs/python/tf/function
[4] https://www.tensorflow.org/federated/api_docs/python/tff/tf_computation
[5] https://www.tensorflow.org/federated/api_docs/python/tff/federated_broadcast

and `tff.federated_mean`[6] inside a `tff.federated_computation`

3. An outer driver script that simulates the control logic of a production FL system, selecting simulated clients from a dataset and then executing federated computations defined in 2).

For the basic `FedAvg` algorithm we use the `tff.learning.build_federated_averaging_process` built-in function. It uses the `model_fn` function to initialize the server and clients models, the `client_optimizer_fn` to update the clients and `server_optimizer_fn` to update the server. Then, an initial state is instantiated and updated at each round. The `FedAvg` iterative process is shown in Listing 3.1.

```
import tensorflow as tf
import tensorflow_federated as tff

fedavg_iter_process = tff.learning.
   build_federated_averaging_process(
        model_fn,
        client_optimizer_fn,
        server_optimizer_fn)

state = fedavg_iter_process.initialize()

for round in range(NUM_ROUNDS):
    state = fedavg_iter_process.next(state,
   training_dataset)
```

Listing 3.1: Example code of the TFF iterative process for `FedAvg`.

Concerning the `FedCurv` algorithm, there isn't a build-in function in TFF, so the iterative process has been done from scratch, that is shown in Listing 3.2.

```
import tensorflow as tf
import tensorflow_federated as tff

fedcurv_iter_process = tff.templates.IterativeProcess(
```

[6] https://www.tensorflow.org/federated/api_docs/python/tff/federated_mean

```
5         initialize_fn=server_init_tff,
6         next_fn=next_fn)
7
8 state = fedcurv_iter_process.initialize()
9
10 for round in range(NUM_ROUNDS):
11     state = fedcurv_iter_process.next(state,
    training_dataset)
```

Listing 3.2: Example code of the TFF iterative process for `FedCurv`.

In the `next_fn` function are executed the `ServerUpdate` and `ClientUpdate` functions of the Alg. 2. In Listing 3.3 there is the implementation of the `next_fn` that is the central point of the parameters dispatching from server to clients and vice versa. In the `next_fn` the server parameters are broadcasted to each clients and used to update the clients by using the `client_update_fn` function; then the clients parameters are collected and aggregated and send to the server for updating by the use of `server_update_fn`.

```
1 @tff.federated_computation(federated_server_type,
    federated_dataset_type)
2   def next_fn(server_state, federated_dataset):
3     # Broadcast the server weights to the clients.
4     server_state_at_client = tff.federated_broadcast(
    server_state)
5     server_weights_at_client = server_state_at_client.
    trainable_weights
6     u_global_at_client = server_state_at_client.u_global
7     v_global_at_client = server_state_at_client.v_global
8     # Local client update.
9     model_deltas, u_locals, v_locals = tff.federated_map(
10         client_update_fn,
11         (federated_dataset, server_weights_at_client,
    u_global_at_client, v_global_at_client))
12     # Client-to-server upload and aggregation.
13     mean_model_delta = tff.federated_mean(model_deltas)
14     u_sum = tff.federated_sum(u_locals)
15     v_sum = tff.federated_sum(v_locals)
16     # Server update.
17     server_state = tff.federated_map(
18         server_update_fn, (server_state, mean_model_delta
```

```
    , u_sum, v_sum))
19    return server_state
```

Listing 3.3: TFF implementation of the next_fn of `FedCurv`.

The `client_update_fn` and `server_update_fn` are shown, respectively, in Listing 3.4 and 3.5.

```
1 @tf.function
2   def client_update(model, dataset, server_weights,
    server_u, server_v):
3     fish_tmp = [tf.zeros_like(tensor) for tensor in model
    .trainable_variables]
4     u_local = [tf.zeros_like(tensor) for tensor in
    server_u]
5     v_local = [tf.zeros_like(tf.multiply(f, w)) for f,w
    in zip(fish_tmp, server_v)]
6     client_weights_old = [tf.zeros_like(tensor) for
    tensor in model.trainable_variables]
7     client_weights_new = [tf.zeros_like(tensor) for
    tensor in model.trainable_variables]
8     updated_weights = [tf.zeros_like(tensor) for tensor
    in model.trainable_variables]
9
10    client_weights_old = model.trainable_variables
11    client_weights_new = model.trainable_variables
12    updated_weights = model.trainable_variables
13
14    for batch in iter(dataset):
15      fish_diag = fisher_matrix(model, batch)
16      u_local = [fim for fim in fish_diag]
17      v_local = [tf.multiply(fim, w_old) for fim, w_old
    in zip(fish_diag, client_weights_old)]
18      penalty = compute_penalty(batch, client_weights_new
    , server_u, server_v, u_local, v_local)
19
20      with tf.GradientTape() as tape:
21        outputs = model.forward_pass(batch)
22        loss = outputs.loss + penalty
23        grads = tape.gradient(loss, model.
    trainable_variables)
24        optimizer_state, updated_weights =
    client_optimizer.next(optimizer_state,
```

```
    client_weights_new, grads)

        client_weights_old = client_weights_new
        client_weights_new = updated_weights
      return tf.nest.map_structure(tf.subtract,
    client_weights_new, server_weights), u_local, v_local
```

Listing 3.4: TFF implementation of the client_update_fn of `FedCurv`.

```
@tf.function
  def server_update(server_state, mean_model_delta, u_sum
    , v_sum):
      negative_weights_delta = tf.nest.map_structure(lambda
     w: -1.0 * w, mean_model_delta)
      new_optimizer_state, updated_weights =
    server_optimizer.next(
          server_state.optimizer_state, server_state.
    trainable_weights,
          negative_weights_delta)

      return tff.structure.update_struct(
          server_state,
          model=tff.learning.ModelWeights(
              trainable=updated_weights,
              non_trainable=server_state.model.
    non_trainable
          ),
          trainable_weights=updated_weights,
          optimizer_state=new_optimizer_state,
          u_global=u_sum,
          v_global=v_sum)
```

Listing 3.5: TFF implementation of the server_update_fn of `FedCurv`.

The two essential components of `FedCurv` are the computation of the diagonal of the Fisher Information matrix and the computation of the penalty term. Concerning the computation of the Fisher Information matrix, this is shown in Listing 3.6.

```
import tensorflow as tf
import tensorflow_federated as tff

def fisher_diag(model, batch):
```

```
  weights = model.trainable_variables
  n_samples = len(batch)
  fisher_diagonal = [tf.zeros_like(tensor) for tensor in
   weights]
  with tf.GradientTape() as tape:
    output = model.forward_pass(batch).predictions
    log_likelihood = tf.nn.log_softmax(output)
    gradients = tape.gradient(log_likelihood, weights)
  fisher_diagonal = [(grad**2)/n_samples for grad in
   gradients]
  return fisher_diagonal
```

Listing 3.6: Code for the computation of the diagonal of the Fisher Information matrix.

Instead, the computation of the penalty term is shown in Listing 3.7.

```
import tensorflow as tf
import tensorflow_federated as tff

def compute_penalty(weights, u_global, v_global, u_local,
    v_local):
  penalty = 0.
  for w, u_g, v_g, u_l, v_l in zip(weights, u_global,
   v_global, u_local, v_local):
      u = u_g - u_l
      v = v_g - v_l
      penalty += tf.reduce_sum((w**2)*u - 2*w*v)
  return 1e-4*penalty
```

Listing 3.7: Code for the computation of the penalty term.

Chapter 4

Discriminator for Anomalous Client detection

A common need when analyzing real-world data-sets is determining which instances stand out as being dissimilar to all others. Such instances are known as *anomalies*, and the goal of *anomaly detection* (also known as outlier detection) is to determine all such instances in a data-driven fashion [23]. Anomalies can be caused by errors in the data but sometimes are indicative of a new, previously unknown, underlying process. In the broader field of machine learning, the recent years have witnessed a proliferation of deep neural networks, with unprecedented results across various application domains. Deep learning (DL) is a subset of ML that achieves good performance and flexibility by learning to represent the data as a nested hierarchy of concepts within layers of the neural network.

Anomalies are also referred to as abnormalities, deviants, or outliers in the data mining and statistics literature [24]. Despite the substantial advances made by DL methods in many ML problems, there is a relative scarcity of DL approaches for anomaly detection [25].

Anomalies can be broadly classified into three types: *point anomalies*, *contextual anomalies* and *collective anomalies*.

1. *Point anomaly*: represents a point which stands on a low-density region of the observed distribution;

2. *Contextual anomaly*: represents an observation which is not a point anomaly but it is anomalous with respect to the surrounding context in the sequence of observations;

3. *Collective anomalies*: are anomalies that arise from the observations of the distributions across multiple data streams.

The type of anomaly considered in this Work refers to *point anomalies*. Specifically, we want to detect anomalous client at a certain time. This detection has to be done server side, due to privacy concerns and low computational power at edge devices.

There are a lot of anomaly detection topics like: intrusion detection, fraud detection, banking fraud, malware detection, IoT Big Data anomaly detection, industrial anomalies detection, etc [25]. Among them, we can't find something related to "anomalous machine learning model" detection.

In this chapter is described the approach used to identify anomalous clients during the FL process. The scope is to detect/discriminate and discard those clients, in order to not considering them during the process of an FL algorithm. In this way the aggregated model will not be affected by anomalous clients.

Our approach is to train a simple feedforward neural network on the gradients of clients computed at each backpropagation step, in order to distinguish between: client trained on noisy input and client trained on plain input. The reason of using the clients' gradients comes from the fact that those are the only things that clients share with the server. The gradient taken into account is computed making a subtraction between the weights at time t and at time $t-1$. Concerning the implementation with Tensorflow-Federated, the discriminator is added into the `client_update_fn` and following the same structure and components described in Chapter 3.

The discriminator model is made with the Tensorflow[1] framework with two `Dense`[2] layers, two `Dropout`[3] layer (one after each `Dense` layer) and a `Dense` output layer with `Sigmoid` activation function.

The `FedAvg` and `FedCurv` algorithms with the discriminator (`Disc`) are described, respectively, in the Alg. 3 and 4.

Algorithm 3 `FedAvg+Disc`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

ServerUpdate
  $w_0 \leftarrow$ initialize()
  $disc \leftarrow$ (load `Disc` model)
  for `each round` $t = 1, 2, \ldots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for `each client` $k \in S_t$ **do**
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
      $valid \leftarrow disc(w_t^k - w_{t+1}^k)$
      if not $valid$ **then** $w_{t+1}^k \leftarrow w_t^k$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w)
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for i `from 1 to` E **do**
    for batch $b \in \mathcal{B}$ **do**
      $w \leftarrow w - \eta \nabla \ell(w; b)$
  return w

[1] https://www.tensorflow.org/
[2] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense
[3] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout

Algorithm 4 `FedCurv+Disc`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate. Each client has the reservoir parameter $\mathbf{W}_{in}$ and $\widehat{\mathbf{W}}$, α is the leaky rate, N_y is the output dimension and N_x is the number of reservoir neurons.

ServerUpdate
  let $w_0, u_0, v_0 \in \mathbb{R}^{N_y \times N_x}$
  for `each round` $t = 1, 2, \dots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for `each client` $k \in S_t$ **do**
      $w_{t+1}^k, u_{t+1}^k, v_{t+1}^k \leftarrow \text{ClientUpdate}(k,\, w_t,\, u_t,\, v_t)$
      $valid \leftarrow disc(w_t^k - w_{t+1}^k)$
      if not $valid$ **then** $w_{t+1}^k \leftarrow w_t^k$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$
    $u_{t+1} \leftarrow \sum_{k=1}^{K} u_{t+1}^k$
    $v_{t+1} \leftarrow \sum_{k=1}^{K} v_{t+1}^k$

ClientUpdate$(k,\, w,\, u,\, v)$
  let $u_{local}, v_{local} \in \mathbb{R}^{N_y \times N_x}$
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for i `from 1 to` E **do**
    for batch $b \in \mathcal{B}$ **do**
      $\mathbf{x}(t) = (1-\alpha)\mathbf{x}(t-1) + \alpha \tanh(\mathbf{W}_{in} b + \widehat{\mathbf{W}}\mathbf{x}(t-1))$
      $\mathbf{y}(t) \leftarrow sigmoid(w\mathbf{x}(t))$
      $\nabla\mathcal{LL}_b \leftarrow \frac{\partial log(softmax(\mathbf{y}(t)))}{\partial w}$ ▷ Log-likelihood gradient
      $\mathcal{D}_b \leftarrow \frac{(\nabla\mathcal{LL}_b)^2}{length(b)}$ ▷ Diagonal of the Fisher matrix
      $u_b \leftarrow \mathcal{D}_b$
      $v_b \leftarrow \mathcal{D}_b \times w$
      $u_{local} \leftarrow u - u_b$
      $v_{local} \leftarrow v - v_b$
      $\lambda \leftarrow sum(w^2 \times u_{local} - 2 \times w \times v_{local})$ ▷ Penalty term
      $w \leftarrow w - \eta \nabla[\ell(w; b) + \lambda]$
  return w, u_{local}, v_{local}

Chapter 5

Data Preparation

In this Chapter are described the WESAD Dataset [26] for stress prediction in Section 5.1, and the pre-processing phase in Section 5.2 in order to increasing the ESNs stress prediction performances.

5.1 WESAD Dataset

Affect recognition aims to detect a person's affective state based on observables, with the goal to improve human-computer interaction. WESAD is a publicly available dataset for wearable stress and affect detection. This multimodal dataset features physiological and motion data, recorded from both a wrist/chest-worn device, of 15 subjects during a lab study. For our purposes, we consider the chest device data, and specifically the Electrodermal Activity (EDA), sampled at 4HZ, and the Blood Volume Pulse (BVP), sampled at 64 Hz. The signals are associated to specific label:

- Baseline
- Stress
- Amusement
- Meditation

We want to solve the binary classification problem of predicting if a user is stressed or not. So, the signals associated to labels different from *Stress*, are considered as "Not Stressed" (associated to the integer

value 0), the other signals associated to the label *Stress*, obviously, are considered as "Stressed" (associated to the integer value 1).

5.2 Data Preprocessing

The Data pre-processing phase starts from EDA and BVP signals of the 15 subjects. In the initial phase it is used the NeuroKit2 library[1], a Python Toolbox for Neurophysiological Signal Processing. The EDA signals are cleaned using the function `eda_clean`[2]. The BVP signal, instead, could be considered similar to Photoplethysmogram (PPG) signal (used to detect blood volume changes in the microvascular bed of tissue), and for this reason it is cleaned using the function `ppg_clean`[3].

In the second phase, from the EDA signals are extracted another two features: EDA Thonic (tEDA) and EDA Phasic (pEDA); this is done using the function `eda_phasic`[4]. Consequently, the BVP signals are used to find the PPG peaks by using the function `ppg_findpeaks`[5]. The peaks are used to extract the Heart Rate (HR), using the function `ppg_rate`[6]. The peaks are used also to compute the Heart Rate Variability (HRV), that is done using the function `hrv_time`[7] applied to a sliding window on the PPG peaks signal. The sliding windows considered are of the length of: 5, 25 and 50, and it is multiplied by 64 (the sampling frequency of BVP/PPG) in order to consider a timing of seconds.

[1] https://github.com/neuropsychology/NeuroKit

[2] https://neuropsychology.github.io/NeuroKit/functions/eda.html#neurokit2.eda.eda_clean

[3] https://neuropsychology.github.io/NeuroKit/functions/ppg.html#neurokit2.ppg.ppg_clean

[4] https://neuropsychology.github.io/NeuroKit/functions/eda.html#neurokit2.eda.eda_phasic

[5] https://neuropsychology.github.io/NeuroKit/functions/ppg.html#neurokit2.ppg.ppg_findpeaks

[6] https://neuropsychology.github.io/NeuroKit/functions/signal.html#signal-rate

[7] https://neuropsychology.github.io/NeuroKit/functions/hrv.html#neurokit2.hrv.hrv_time

The result of the previous phases, results in a definition of 3 different dataset (one for each sliding window) with the following fields for each subject:

1. tEDA: EDA Thonic
2. pEDA: EDA Phasic
3. HR: Heart Rate
4. HRV: Heart Rate Variability (sliding window $\in \{5, 25, 50\}$)
5. Label: Stressed or Not Stressed (1 or 0).

The statistics of the dataset after the second phase are shown in Table 5.1.

Feature	Min	Max	Mean	Var
pEDA	-4.4867	5.8884	-0.0140	0.9325
tEDA	0.1639	9.6628	1.8536	4.3981
HR	0.4868	192.0000	77.7142	770.1333
HRV	12.9829	386.3441	45.8408	3245.0130

Table 5.1: Dataset statistics of the extracted features with a sliding window of 25×64. Percentage of samples: label "0": 88.53%, label "1": 11.47%

The third phase consists in resampling the signals with respect to the shortest one for each subject, this is done using the function `signal_resample`[8]. Next, the signals of each subject are standardized with respect to its own baseline, in order to recenter the signals in an interval more appropriate for the ESN, this is done using the `StandardScaler`[9] of the `sklearn`[10] library.

[8] https://neuropsychology.github.io/NeuroKit/functions/signal.html#neurokit2.signal_resample

[9] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[10] https://scikit-learn.org

The statistics of the standardized dataset after the third phase are shown in Table 5.2.

Feature	Min	Max	Mean	Var
pEDA	-46.4578	134.4446	0.9768	133.8697
tEDA	-11.2772	67.8104	4.6638	139.6957
HR	-4.9910	7.4297	0.1300	1.6040
HRV	-3.3037	92.1442	3.9575	108.4837

Table 5.2: Dataset statistics of the standardized features with a sliding window of 25×64. Percentage of samples: label "0": 88.57%, label "1": 11.43%

The last phase is a normalization of the features with an online moving average. The statistics of the normalized dataset are shown in Table 5.3.

Feature	Min	Max	Mean	Var
pEDA	-2.9826	6.8522	0.6837	2.4992
tEDA	-20053.9745	12748.3417	0.6757	8039.7497
HR	-278.8295	537.7899	1.1065	264.5812
HRV	-4.5778	21.5005	1.1469	7.9482

Table 5.3: Dataset statistics of the normalized features with moving average with a sliding window of 25×64. Percentage of samples: label "0": 88.57%, label "1": 11.43%

The last dataset contains $\sim$ 6000 samples for each subject and it is the one used for the Experimental phase in Chapter 6.

Chapter 6

Experiments

In this Chapter we are going to describe the hyper-parameter search and the final structure of the best ESN on the stress prediction task. Next, we will show the settings of the ESN model, found in the previous phase, applied to the FL scenario; these phases are described in Section 6.1. In Section 6.2 are shown the results concerning the FL scenarios (Alg. 1 and 2) and the FL scenarios with the use of a discriminator for anomalous client detection (Alg. 3 and 4).

6.1 Experimental Setup

The experiments are conducted using the Python programming language and the Machine Learning models are written using the Tensorflow framework with the Keras backend. The *reservoir* of the ESN has been implemented using the Tensorflow Addons[1] library, and specifically the `ESN`[2] layer. The *readout* of the ESN has been implemented using a simple `Dense`[3] layer with `Sigmoid`[4] activation function and `class_weight={0: 12, 1: 88}` in order to takle the problem of unbalanced labels distribution (see Table 5.3). The model is compiled

[1] https://www.tensorflow.org/addons

[2] https://www.tensorflow.org/addons/api_docs/python/tfa/layers/ESN

[3] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

[4] https://www.tensorflow.org/api_docs/python/tf/keras/activations/sigmoid

using the `Adam` optimizer[5] and `BinaryCrossentropy`[6] loss function.

The *model selection* phase is conducted using the KerasTuner[7] library. Specifically, it is used the `RandomSearch`[8] class that uses a set of `HyperParameters`[9] values to explore the search space in a random way. The dataset has been splitted in *training set* containing 10 subjects, *validation set* containing 4 subjects and *test set* with just one subject. The search is interrupted when all the hypothesis has been exhausted. The hyper-parameter search space is shown in Table 6.1.

Units	$50, 100, 200$
Leaky (α)	$0.5, 0.8, 1.0$
Learning rate (η)	$0.001, 0.01, 0.1$
Window size	$(5 \times 64), (25 \times 64), (50 \times 64)$
Batch size	$20, 50, 100$
Features	(pEDA, tEDA, HR, HRV), (tEDA, pEDA), (HR, HRV), (pEDA)

Table 6.1: ESN hyper-parameter search space.

The best model parameters has been chosen looking at the F1 score (the harmonic mean of the precision and recall) on the *validation set*. The reason why using the F1 score is that it is more suitable for task with unbalanced class distribution. In Table 6.2 are shown the best parameters associated with a validation F1 score of `0.9106`.

[5] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

[6] https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy

[7] https://keras.io/keras_tuner/

[8] https://keras.io/api/keras_tuner/tuners/random/

[9] https://keras.io/api/keras_tuner/hyperparameters/

Units	α	η	ρ	Window	Batch	Features	Epochs
200	1.0	0.1	0.99	25×64	20	pEDA	1

Table 6.2: Best ESN model parameters associated with a validation F1 score of `0.9106`.

A graphical representation of the architecture of the model is shown in Fig 6.1.

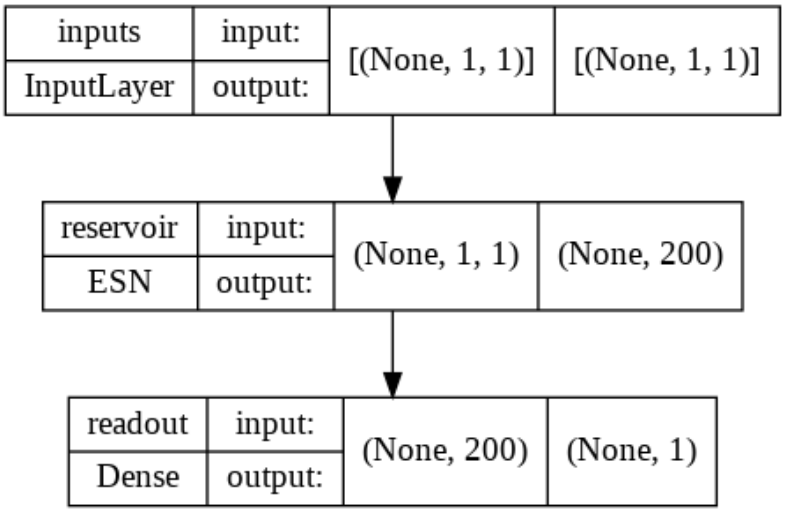

Figure 6.1: ESN model architecture.

In the *model assessment* phase, the ESN has been re-trained on the entire *design set* and having the final performances shown in Table 6.3.

Accuracy (DS/TS)	AUC (DS/TS)	F1 (micro) (DS/TS)
0.8986/0.9018	0.8092/0.9419	0.8986/0.9018

Table 6.3: Best model performances on the *design set* (DS) and *test set* (TS).

After finding the best ESN model, that from now on we will call it *Centralized ESN*, we need to distribute the model, using FL algorithms, in order to see how it performs. The federated learning experiments are conducted using Tensorflow-Federated[10] framework. The experiments

[10] https://www.tensorflow.org/federated

are divided in two phases:

1. *Primal phase*: the splitting of the dataset is the same as before. The *training set* contains 10 subjects, *validation set* contains 4 subjects and *test set* with just one subject.

2. *Discriminative phase*: the splitting of the dataset is a little bit different. The *training set* contains 10 subjects, *validation set* contains 2 subjects and *test set* with just one subject. The 2 subjects (that are not seen before from the *Centralized ESN*) removed from the *validation set* are used to train the discriminator to predict anomalous clients. The 10 clients are setup and divided into: 5 "fake" clients and 5 "real" cients. At each round 5 clients, among reals and fakes, are randomly selected. The discriminator has to discard the fake clients.

In both phases the datasets are wrapped into the `ClientData`[11] class, in order to have a client for each subject. So, during the FL iterative process we have a number of 10 clients in total. Always in both phases, the parameters considered are the following:

- Random number of clients to be considered at each round
- Injected input noise on each client, sampled from a gaussian distribution with a certain standard deviation
- Number of rounds (at each round the model is trained for a single epoch).

In the preliminaries results, shown in Appendix A, it is noticeable that the number of clients per round, the input noise and the number of rounds are not so significant. So, the FL experiments are conducted considering the maximum number of round of `78`, the injected input noise of `0.5` and the number of clients per round of `5`.

In order to carry on the *discriminative phase*, we have to train the discriminator. The Discriminator is a simple feedforward neural network,

[11] https://www.tensorflow.org/federated/api_docs/python/tff/simulation/datasets/ClientData

that accepts an input with size `201` (that corresponds to the dimension of the readout parameters), with two `Dense` layers, two `Dropout`[12] layer and a `Dense` output layer with `Sigmoid` activation function. The discriminator is compiled with `BinaryCrossentropy` loss and `Adam` optimizer with a learning rate of `0.001`. The hyper-parameters have been choosen with a random search. The discriminator architecture is shown in Fig. 6.2.

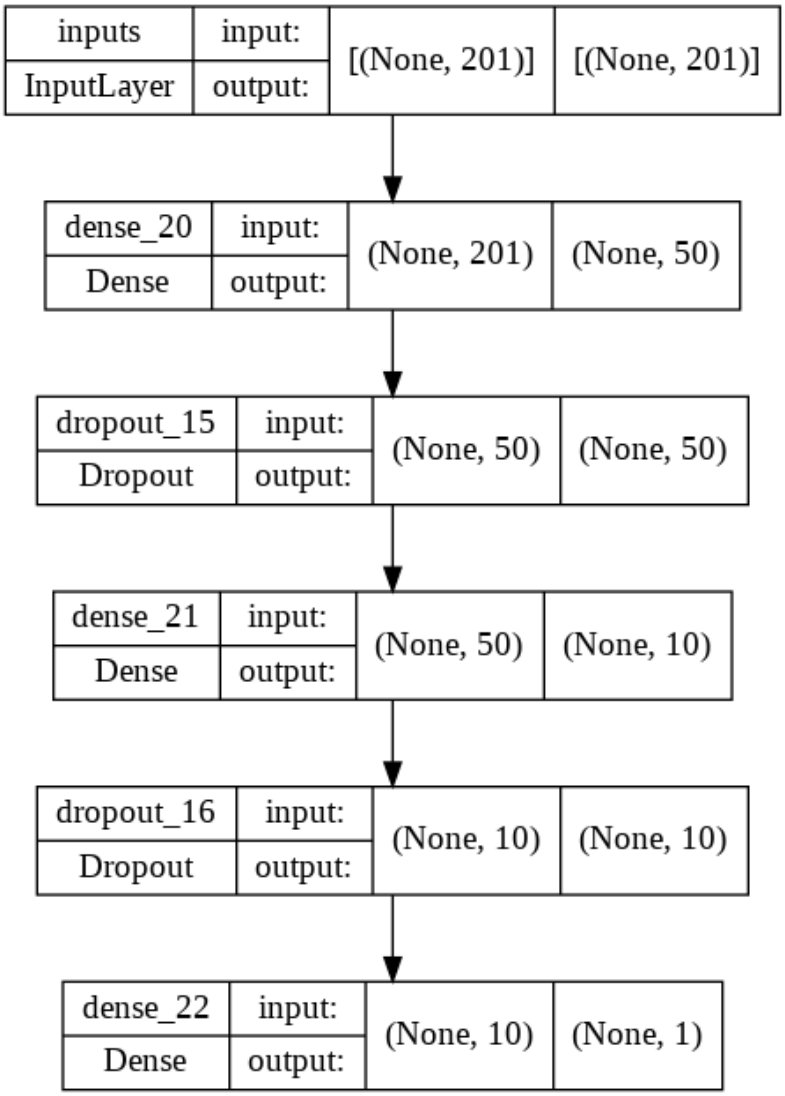

Figure 6.2: Discriminator model architecture.

The discriminator has to solve the problem of classifying if an ESN backpropagation gradient, during the process of stress prediction, is

[12] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout

"real" or "fake". In order to train the discriminator we have to generate the data to give in input. To do so, we have instantiated two random ESN (with the same structure of Fig. 6.1): the *Fake ESN* trained on a noisy data and *Real ESN* trained on the original data. The *training set* is made by the two subject extracted from the *validation set* used before, instead the *test set* is the same used in the previous phases. The fake data noise is generated from a gaussian distribution with standard deviation of `1.0`. The *Fake ESN* reaches a test F1 score of `0.8836` and the *Real ESN* reaches a test F1 score of `0.8839`.

The gradients collected during the training process of the *Real ESN* and *Fake ESN*, are divided in *training set* and *test set*. The *training set* is fed in input to the discriminator and trained for `5` epochs, with a batch size of `64` and a validation split of 20%. The validation binary accuracy, at the end of the process, is `0.9974`, the F1 score on the *training set* is `0.9941` and the F1 score on the *test set* is `0.9925`.

6.2 Experimental Results

Differently from the classical `FedAvg` (Alg. 1) that averages over clients weights, Tensorflow-Federated has a slightly different approach. The deltas of the clients weights after local training are sent back to the server and averaged, in order to compute a pseudo-gradient used to apply standard optimization techniques on server-side [27].

This implementation allow us to setup different learning rates to the server optimizer. In our experiments the server optimizer is `SGD`[13], and we will use the following learning rates: `1.0`, `0.1`, `0.01`, `0.001`. The server with `SGD` with a learning rate of `1.0` corresponds to the classical `FedAvg` algorithm. In Figures 6.3, 6.4, 6.5 and 6.6 are shown the results of the *primal phase* considering `FedAvg` and `FedCurv` with different server learning rate, with clients reservoir different from the server and equal to the server; Are also considered the algorithms applied to the noisy inputs.

[13] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD

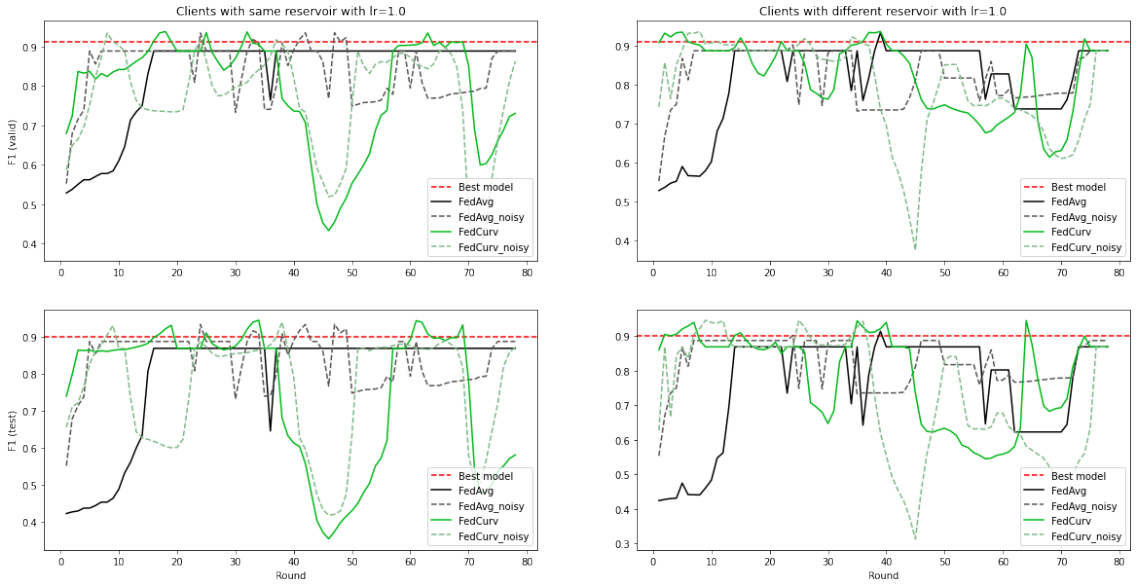

Figure 6.3: Results of the *primal phase* of Federated Learning with a learning rate of `1.0`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

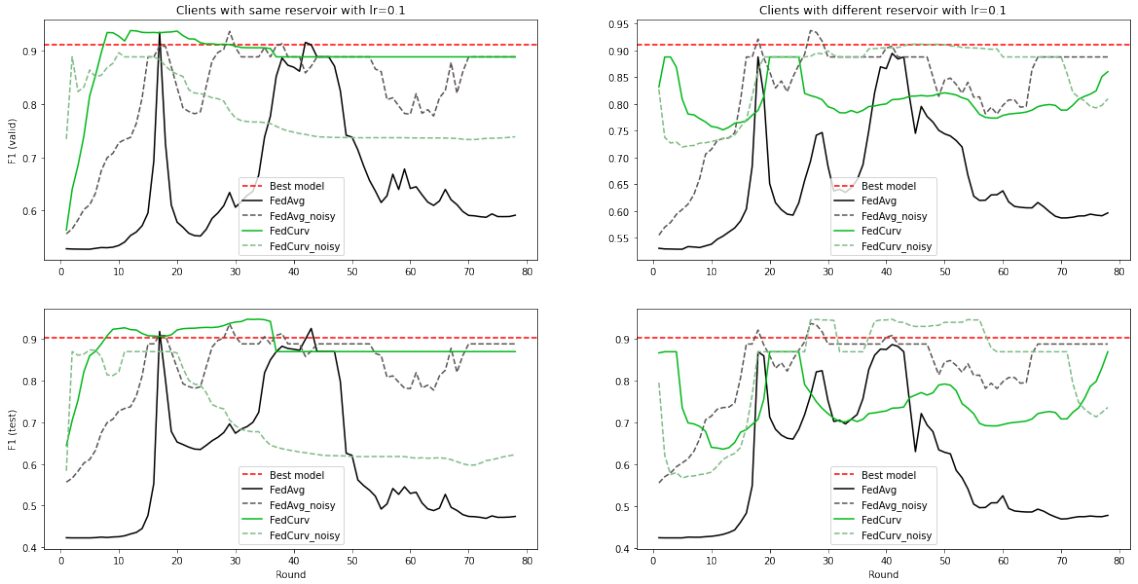

Figure 6.4: Results of the *primal phase* of Federated Learning with a learning rate of `0.1`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

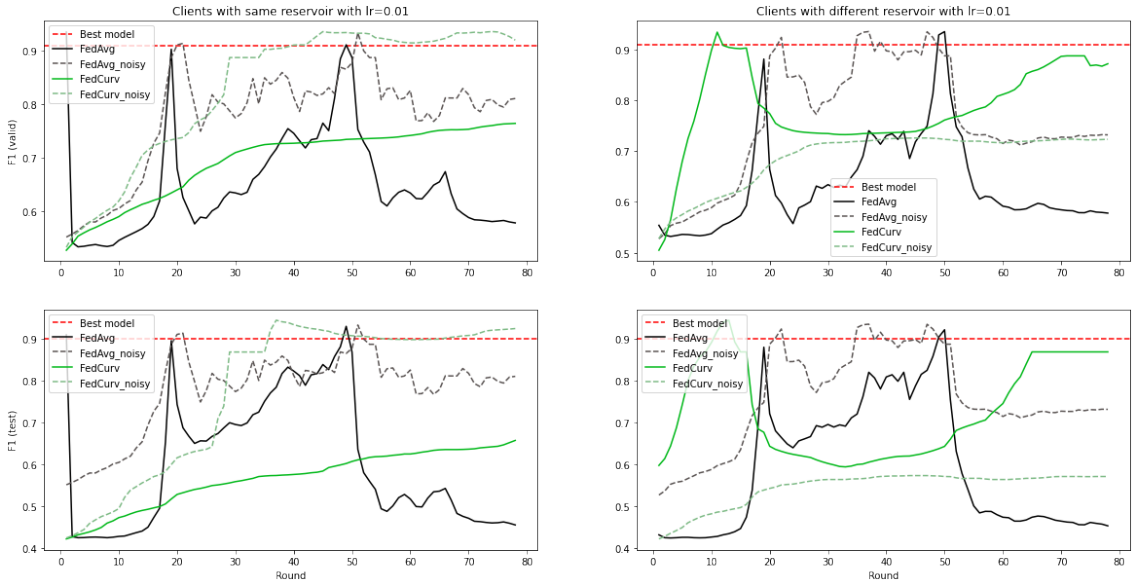

Figure 6.5: Results of the *primal phase* of Federated Learning with a learning rate of `0.01`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

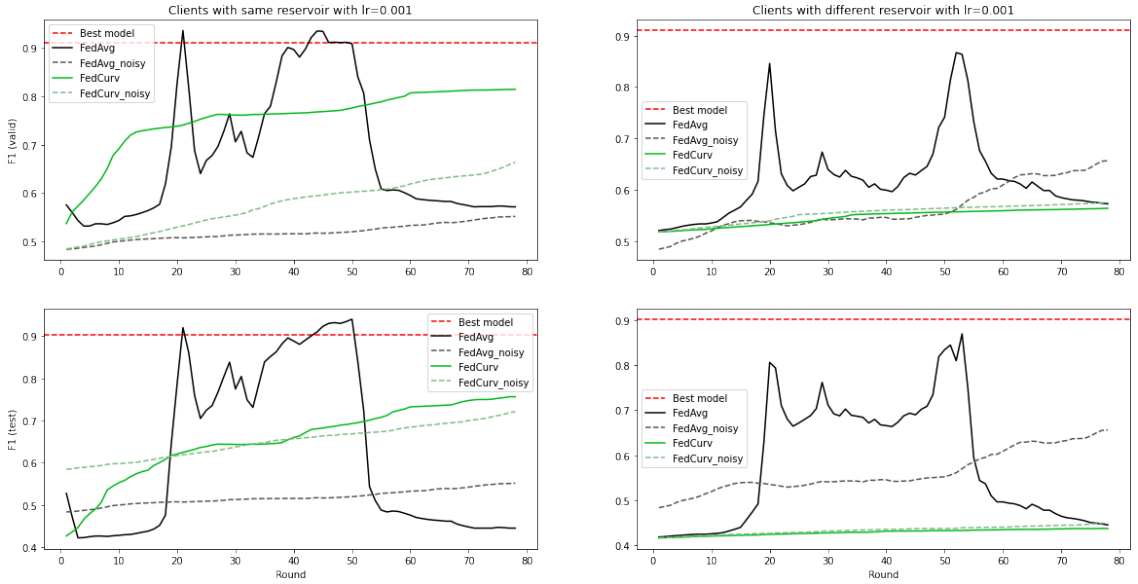

Figure 6.6: Results of the *primal phase* of Federated Learning with a learning rate of `0.001`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

The plots in Figures 6.4, 6.5 and 6.6 highlights the problem of *catastrofic forgetting* in `FedAvg` with learning rates different from `1.0` and `FedCurv` alleviates this problem having more growing monotone curves. In the case of learning rate of `1.0` of Fig. 6.3, the algorithms have similar behaviours.

In Fig. 6.7 is shown a comparison among the best models at the last round with and without the same reservoir. We can see that `FedAvg` and `FedCurv` are similar in both cases, but `FedCurv` is able to overcome the centralized model more times than `FedAvg`. In the case of different reservoirs, we can see an unstable behaviour with respect to the case of clients with the same reservoir.

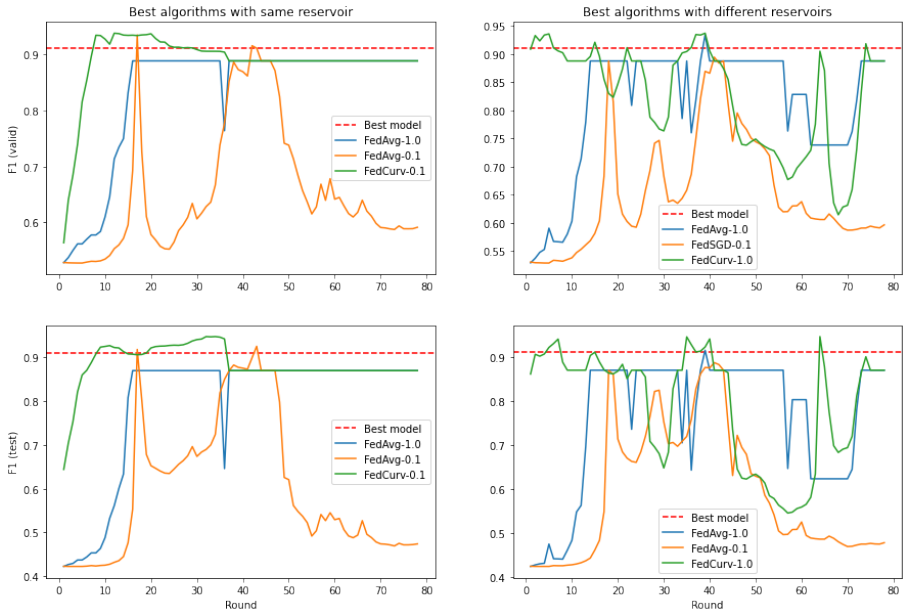

Figure 6.7: Comparison of the best results of the *primal phase* of Federated Learning. 1^{st} row with the scores on validation set, 2^{nd} row on test set. Left column with clients with the same reservoir. Right column with clients with different reservoir.

In Figures 6.8, 6.9, 6.10 and 6.11 are shown the results of the *discriminative phase*, in which at each round if a client is marked as "not valid", the server uses the previous client weight and not the newest.

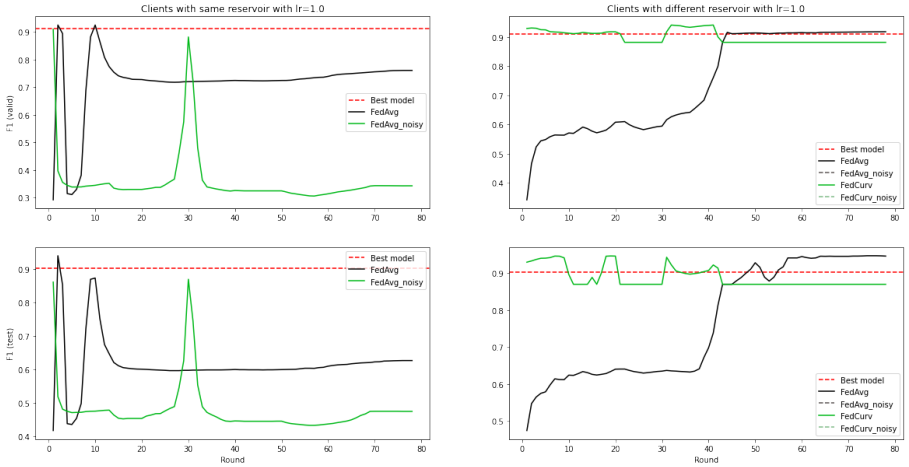

Figure 6.8: Results of the *discriminative phase* of Federated Learning with a learning rate of `1.0`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

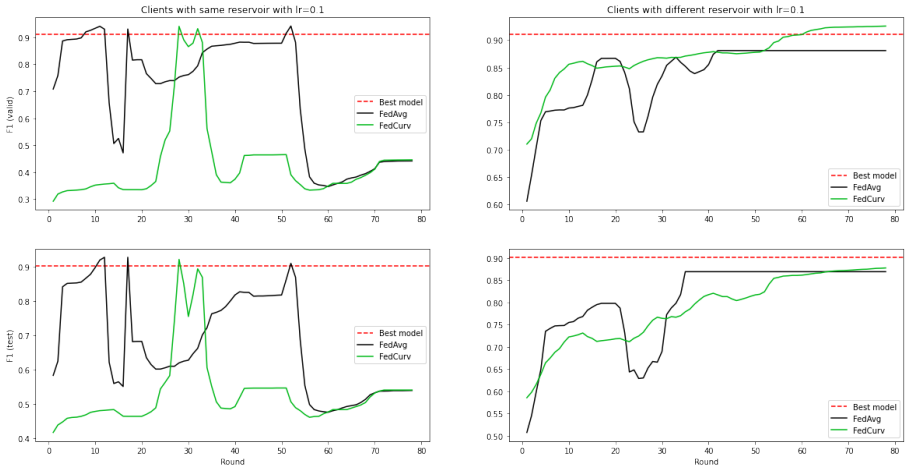

Figure 6.9: Results of the *discriminative phase* of Federated Learning with a learning rate of `0.1`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

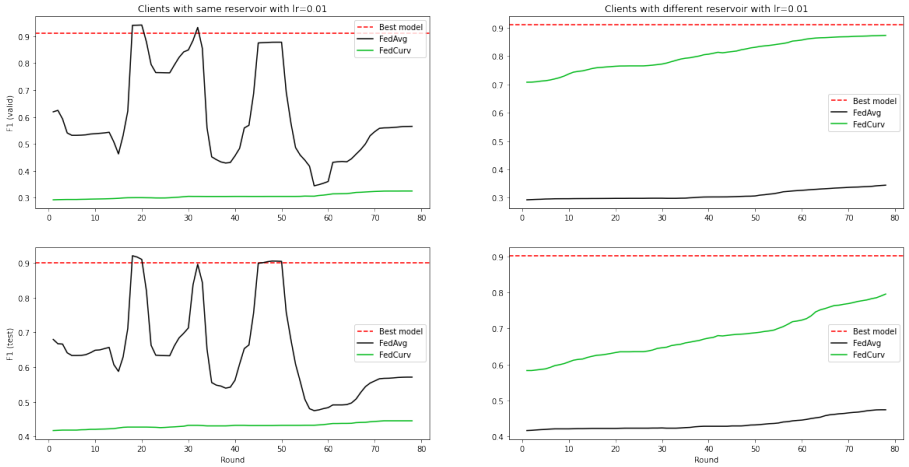

Figure 6.10: Results of the *discriminative phase* of Federated Learning with a learning rate of `0.01`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

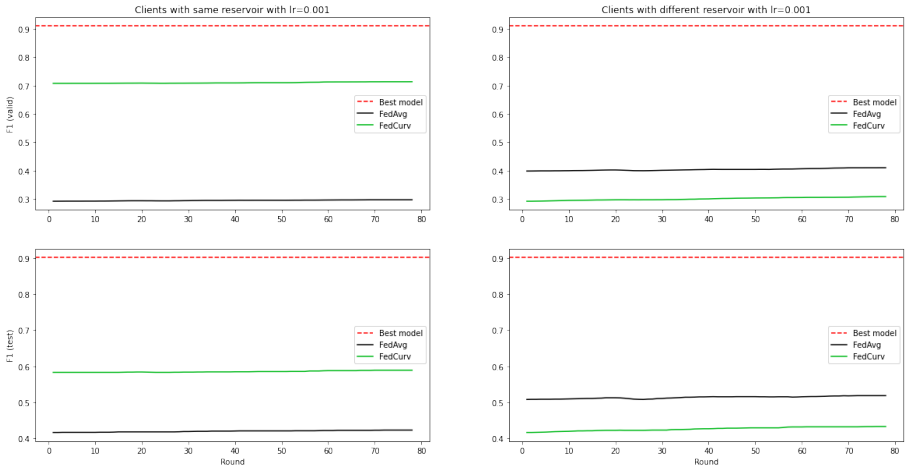

Figure 6.11: Results of the *discriminative phase* of Federated Learning with a learning rate of `0.001`. In the left column the results with clients with the same reservoir. In the right column the results with clients with different reservoir. In the 1^{st} row the scores on the validation set. In the 2^{nd} row the scores on the test set.

The above results shows how the discriminator affect in a positive way the FL algorithms with different reservoir. `FedCurv` is able to match, and some times outperform, the centralized model performances; catastrofic forgetting in `FedAvg` is slightly alleviated. In Fig. 6.12 there is

a comparison of the best models of the *discriminative phase*, in which we can appreciate the benefits of the discriminator in the plots in which the reservoir of the clients are different.

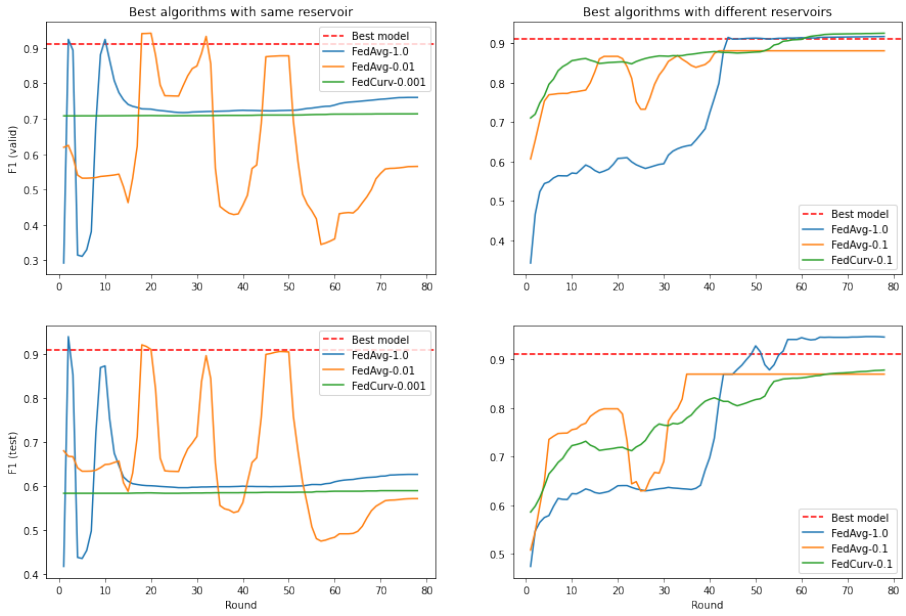

Figure 6.12: Comparison of the best results of the *discriminative phase* of Federated Learning. 1^{st} row with the scores on validation set, 2^{nd} row on test set. Left column with clients with the same reservoir. Right column with clients with different reservoir.

In Table 6.4 are listed the overall performances of the FL algorithm without and with the discriminator with clients having the same reservoir of the server. The SOTA ESN [2] solve a multi-label task, so it is not comparable with our centralized ESN that solves a binary classification task. Since, the accuracy and F1 scores of the centralized model are the same, we can compare it with the SOTA. We can notice that our centralized model overcome SOTA performances, taking in mind that the tasks are different. Looking the results of the *primal phase*, no one of the FL algorithms is able to overcome the centralized model, even if the scores of the best models are close to those of the central model.

ESN Models	Train Acc.	Test Acc.	Train F1	Test F1
`SOTA` (multi-label class.)	0.8378	0.7792	-	-
`Central` (binary class.)	0.8986	0.9018	0.8986	**0.9018**
`FedAvg-1.0`	-	-	0.8878	**0.8693**
`FedAvg-0.1`	-	-	0.5913	0.4741
`FedAvg-0.01`	-	-	0.5784	0.4560
`FedAvg-0.001`	-	-	0.5711	0.4451
`FedCurv-1.0`	-	-	0.7299	0.5813
`FedCurv-0.1`	-	-	0.8878	**0.8693**
`FedCurv-0.01`	-	-	0.7646	0.6580
`FedCurv-0.001`	-	-	0.8140	0.7562
`FedAvg-1.0+Disc`	-	-	0.7601	0.6262
`FedAvg-0.1+Disc`	-	-	0.4415	0.5392
`FedAvg-0.01+Disc`	-	-	0.5652	0.5710
`FedAvg-0.001+Disc`	-	-	0.2972	0.4233
`FedCurv-1.0+Disc`	-	-	0.3427	0.4741
`FedCurv-0.1+Disc`	-	-	0.4456	0.5405
`FedCurv-0.01+Disc`	-	-	0.3251	0.4451
`FedCurv-0.001+Disc`	-	-	0.7136	0.5891

Table 6.4: Overall comparison of the best models with different FL algorithms. The reservoirs of the clients in all the FL algorithms are equal to the server.

In Table 6.5 are listed the overall performances of the FL algorithm with and without the discriminator with clients having different reservoir from the server. We can notice that classical `FedAvg` algorithm in combination with the discriminator is able to exceed the central model performances.

ESN Models	Train Acc.	Test Acc.	Train F1	Test F1
`SOTA` (multi-label class.)	0.8378	0.7792	-	-
`Central` (binary class.)	0.8986	0.9018	0.8986	**0.9018**
`FedAvg-1.0`	-	-	0.8878	0.8693
`FedAvg-0.1`	-	-	0.5965	0.4772
`FedAvg-0.01`	-	-	0.5779	0.4538
`FedAvg-0.001`	-	-	0.5723	0.4457
`FedCurv-1.0`	-	-	0.8878	0.8693
`FedCurv-0.1`	-	-	0.8603	0.8693
`FedCurv-0.01`	-	-	0.8722	0.8693
`FedCurv-0.001`	-	-	0.5636	0.4379
`FedAvg-1.0+Disc`	-	-	0.9172	**0.9457**
`FedAvg-0.1+Disc`	-	-	0.8810	0.8693
`FedAvg-0.01+Disc`	-	-	0.3440	0.4744
`FedAvg-0.001+Disc`	-	-	0.4103	0.5187
`FedCurv-1.0+Disc`	-	-	0.8810	0.8693
`FedCurv-0.1+Disc`	-	-	0.9257	**0.8778**
`FedCurv-0.01+Disc`	-	-	0.8733	0.7955
`FedCurv-0.001+Disc`	-	-	0.3084	0.4332

Table 6.5: Overall comparison of the best models with different FL algorithms. The reservoirs of the clients in all the FL algorithms are different from the server.

In Fig. 6.13 we can see a detail of the discriminative process during rounds of `FedAvg-1.0+Disc` with different reservoir. The discriminator has a Mean Absolute Error of `1.1410`. So at each round, on average, it wrongly recognizes one client as valid.

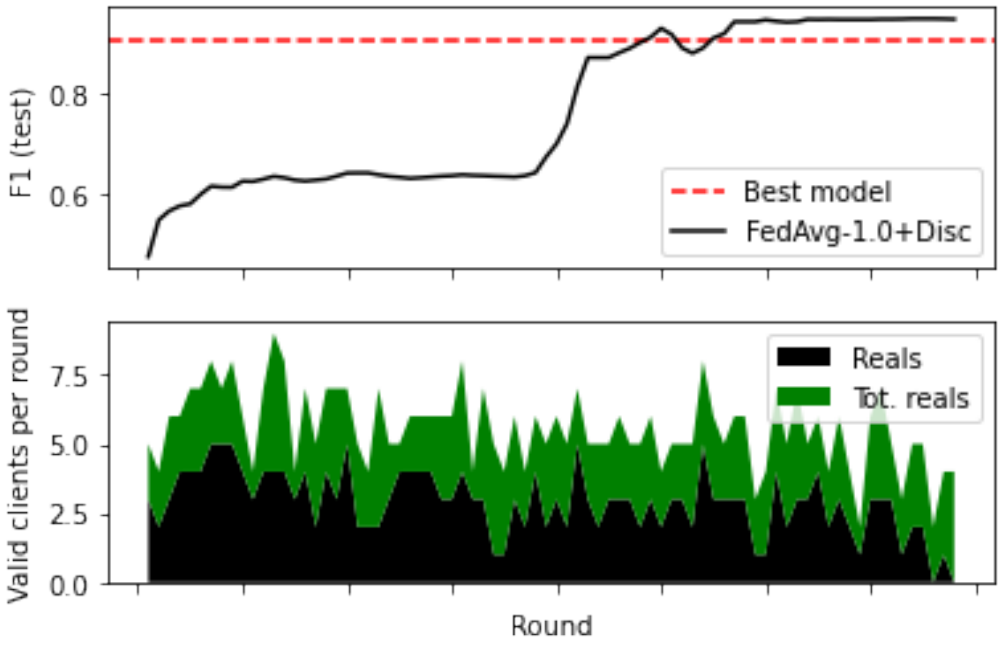

Figure 6.13: At the top part the `FedAvg-1.0+Disc` with different reservoir scores on test set. At the bottom part the stack plot of the `Disc` predictions of valid clients (black) vs real valid clients (green) at each round. The `Disc` predictions Mean Absolute Error is 1.1410.

Chapter 7

Conclusions

Echo State Networks are the state-of-the-art Recurrent Neural Networks for signal processing. It is suitable for a federated setting scenarios because, from the mathematically point of view, the aggregated model is the same of the model learned using the whole dataset [2].

Echo State Networks are also appropriate for tackling the problem of offloading data coming from data collected by edge devices and connection efficiency among network nodes in data exchanges. The only parameters exchanged among nodes are the *readout* weights.

The Echo State Network model trained on the WESAD dataset outperforms the state-of-the-art model, even if they face two slightly different tasks.

In the experimental results the classical `FedAvg` and `FedCurv` with a learning rate of `0.1`, have the same performances, nevertheless, `FedCurv` has more increasing monotone performances then `FedAvg` and so it is more reliable at a certain time step.

The second part of the experimental results shows us how a simple Federated Learning algorithm as `FedAvg` in combination with a simple anomalous client discriminator, is able to overcome the central model performances.

7.1 Future Works

The use of Tensorflow-Federated is not suitable for extremely customized Federated Learning algorithm. The inclusion of the discriminator in the FL algorithms was a little bit tricky. As future work, the creation of a federated learning framework could be done in order to compare and customize different FL algorithms.

In this Work we adapt the `FedCurv` algorithm to ESNs using the federated optimization technique of the Tensorflow-Federated framework.

As future work, we can adapt `FedCurv` to ESNs using closed form solutions for the readout adaptation (as in Bacciu et al. [2]). We developed a preliminary version of this approach, namely `Partial IncFed`, whose details and first experiments are described in Appendix B. The preliminary results shows as the method has a better generalization capability than `IncFed` and contextually reducing the communication cost.

Given the good potential showed by such approach, in the future we aim to explore other methodologies to select the most important neurons and reduce the communication cost. A candidate for such purpose is SVD, which allows to select the principal components of the spaces spanned by A and B, thus reducing their dimensionality.

Appendix A

FL Preliminary Results

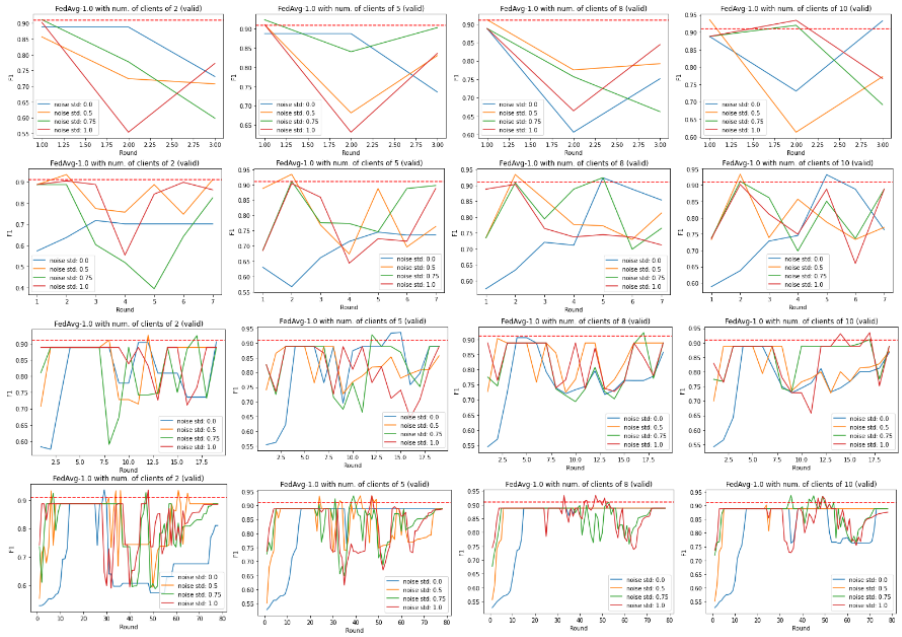

Figure A.1: `FedAvg-1.0` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

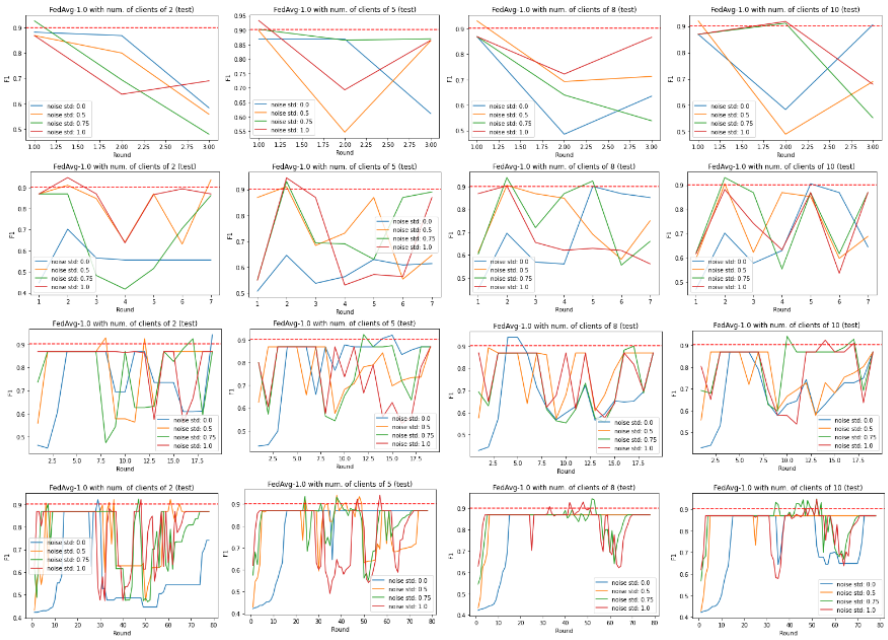

Figure A.2: `FedAvg-1.0` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

Figure A.3: `FedAvg-0.1` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

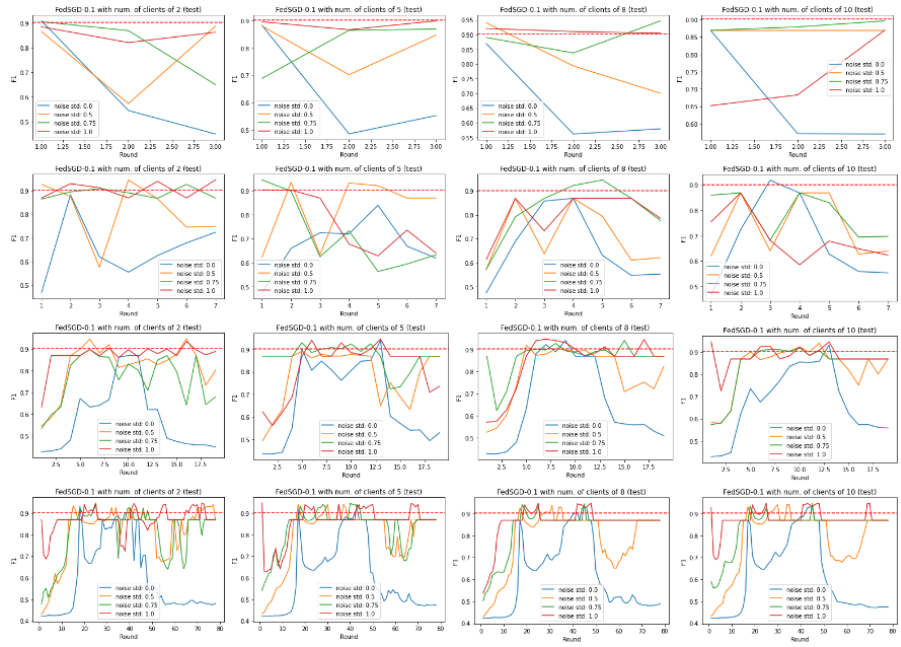

Figure A.4: `FedAvg-0.1` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

Figure A.5: `FedAvg-0.01` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

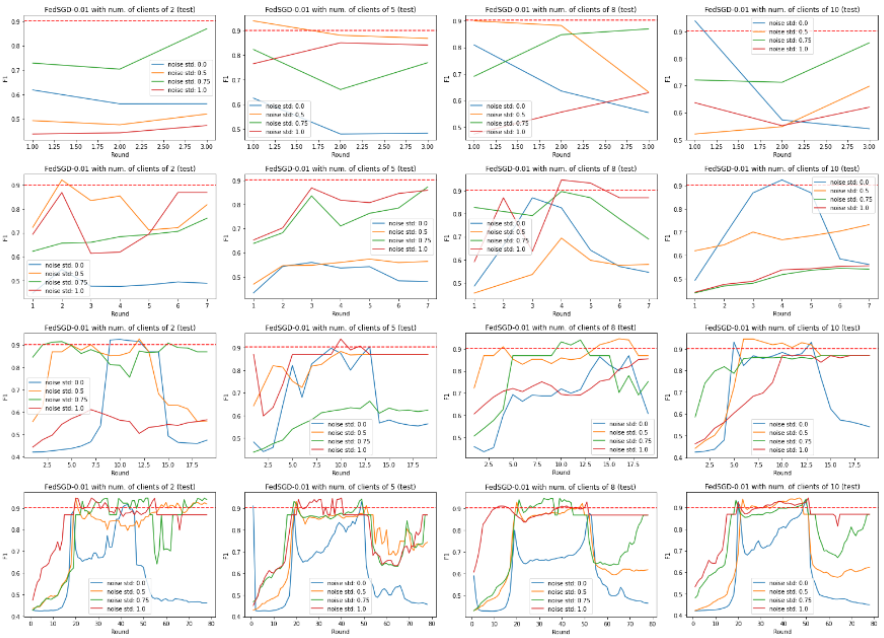

Figure A.6: `FedAvg-0.01` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

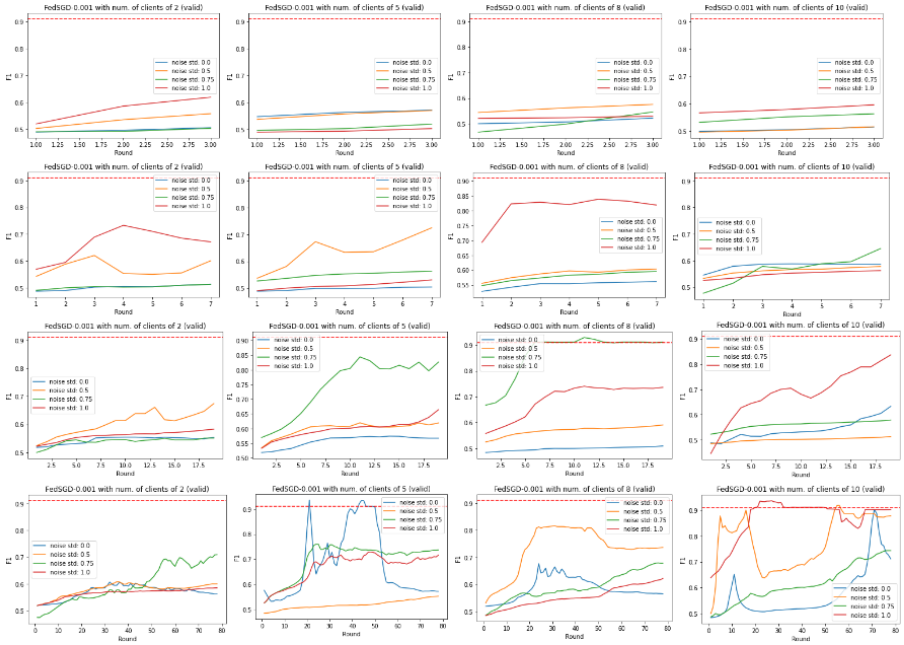

Figure A.7: `FedAvg-0.001` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

Figure A.8: `FedAvg-0.001` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

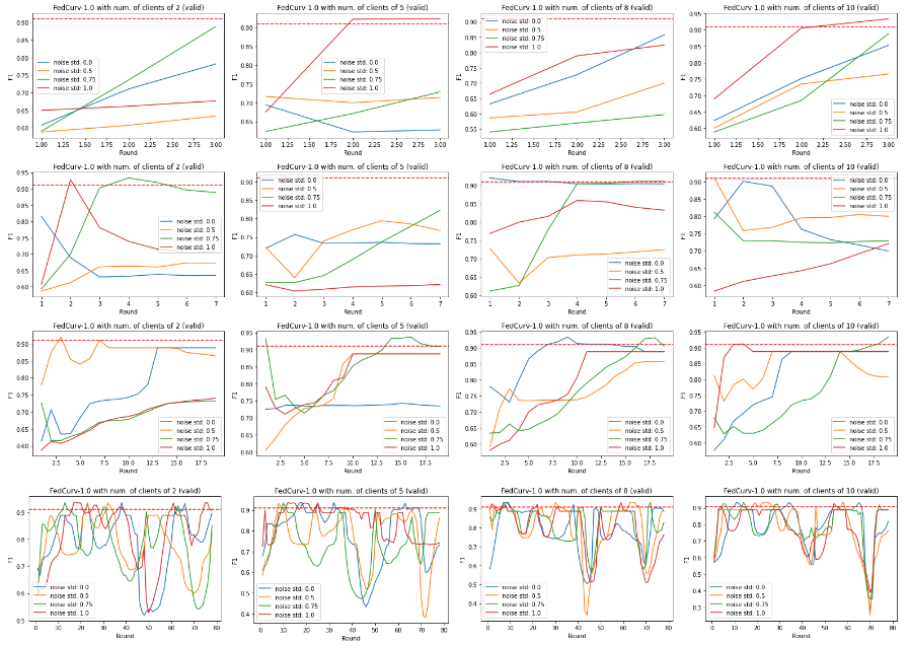

Figure A.9: `FedCurv-1.0` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

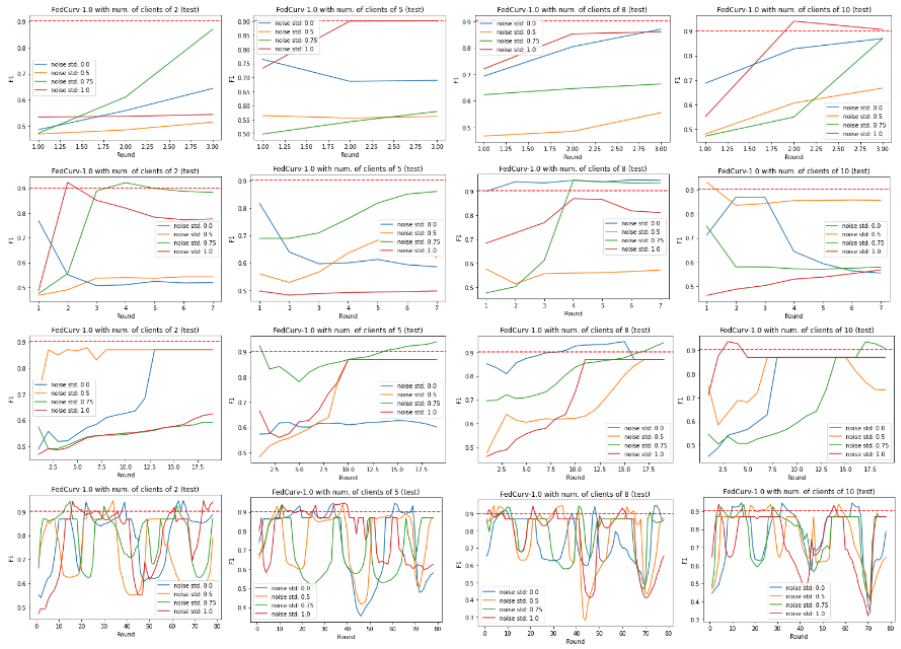

Figure A.10: `FedCurv-1.0` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

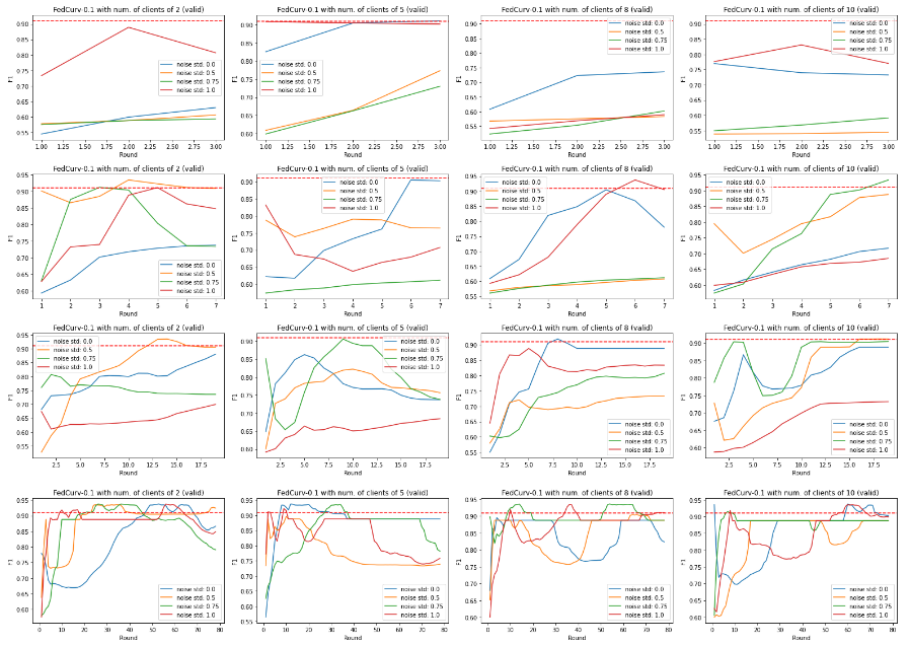

Figure A.11: `FedCurv-0.1` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

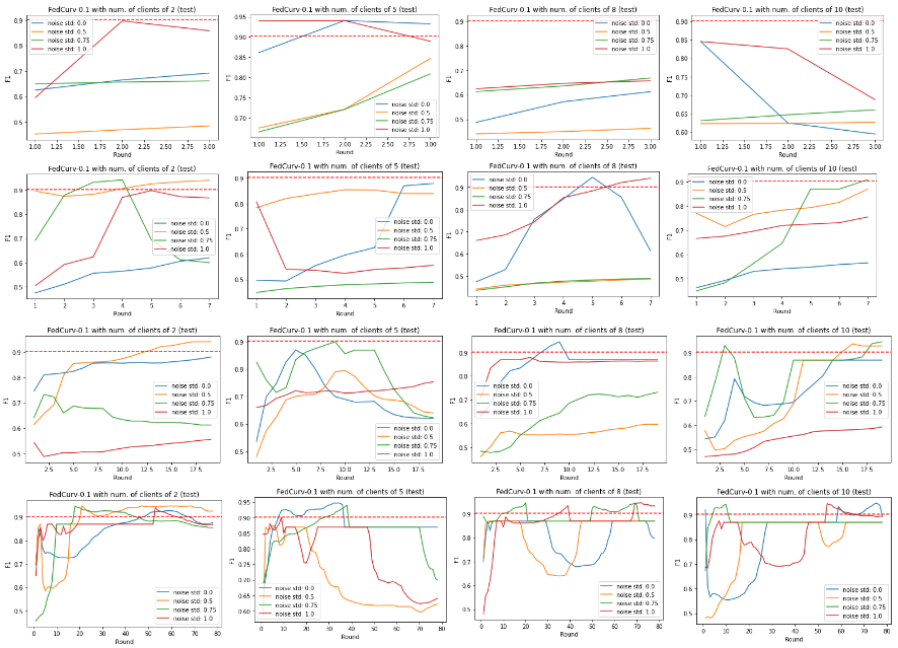

Figure A.12: `FedCurv-0.1` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

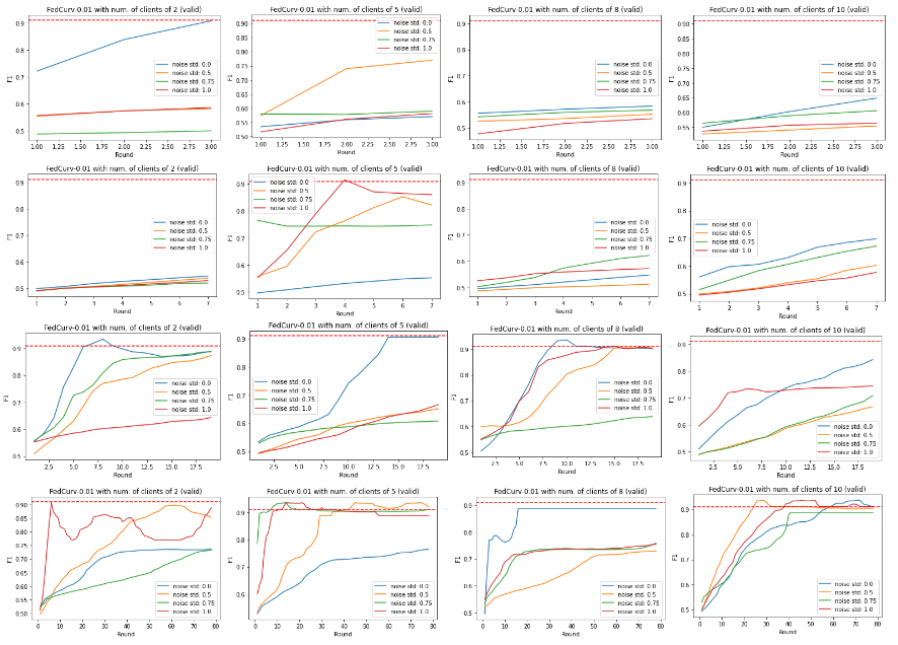

Figure A.13: `FedCurv-0.01` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

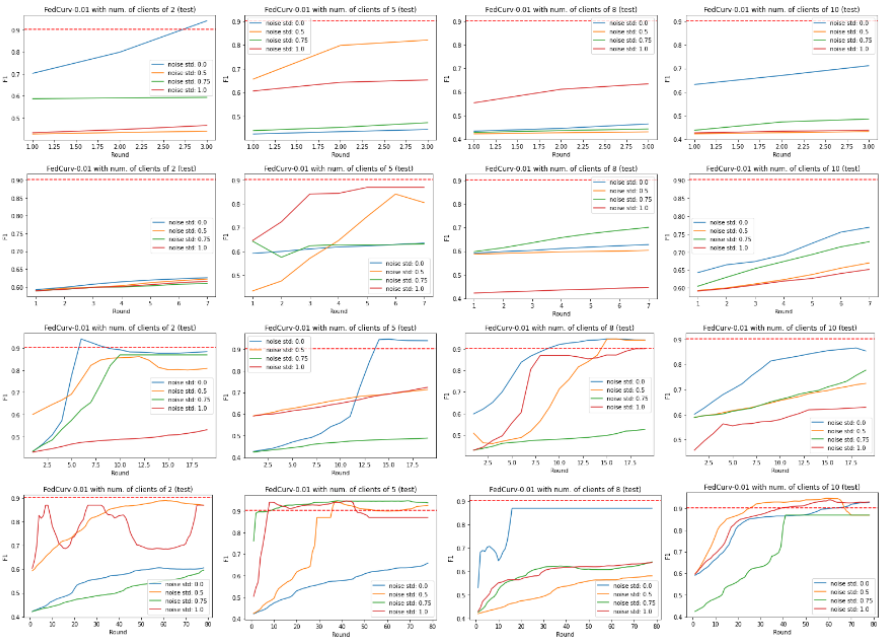

Figure A.14: `FedCurv-0.01` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

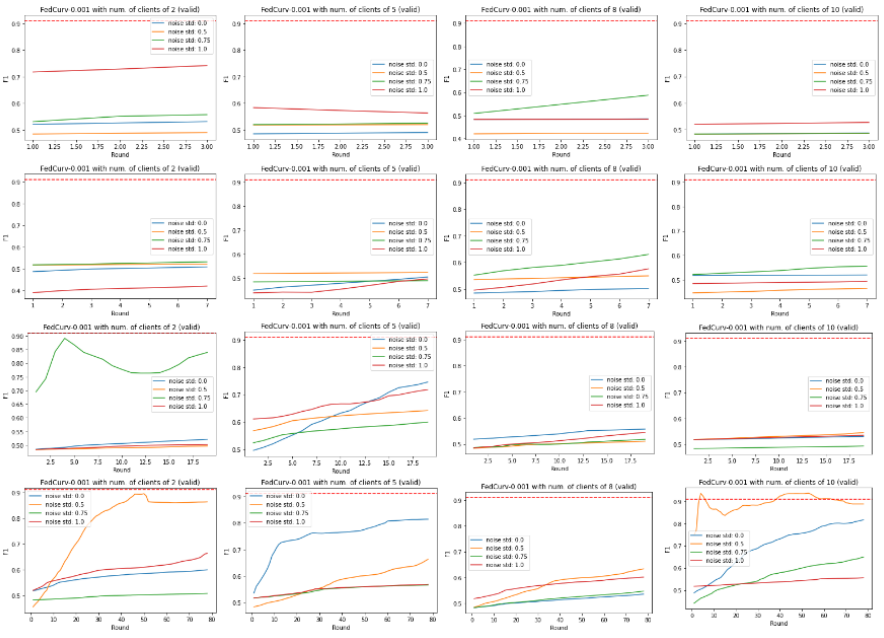

Figure A.15: `FedCurv-0.001` preliminary experiments on *validation set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

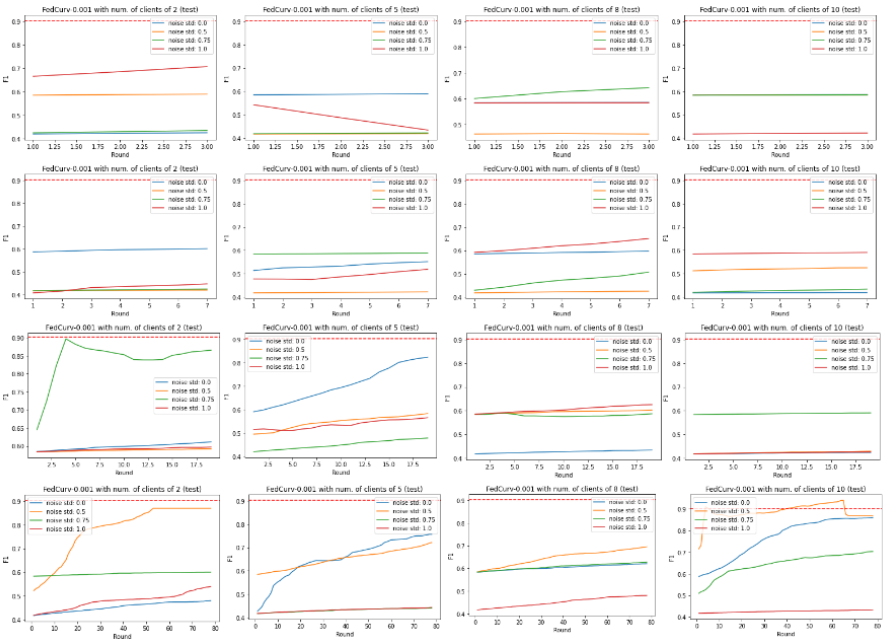

Figure A.16: `FedCurv-0.001` preliminary experiments on *test set*. From top to bottom with increasing number of rounds. From left to right with increasing number of clients per round.

Appendix B

Partial Incremental Federated Learning

In this Chapter are shown the preliminary results of a novel approach called *Partial Incremental Federated Learning* (`Partial IncFed`). This algorithm is an importance-based method that aims to reduce the cost of communication among clients and server in a federated learning scenario.

B.1 Incremental Federated Learning

Incremental Federated Learning (`IncFed`) algorithm exploits the training characteristics of ESNs to obtain an optimal form of federated learning.

In the end-to-end training scheme of an ESN, the input sequences from the training dataset are fed to the reservoir. Then, the relevant states on which the network must learn to perform predictions are collected column-wise into a matrix $\mathbf{S} \in \mathbb{R}^{N_x \times N_{train}}$, where N_{train} is the number of such states, and the associated targets are collected into the matrix $\mathbf{Y} \in \mathbb{R}^{N_y \times N_{train}}$. Finally, the matrix $\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$ is obtained as the solution to a least squares minimization problem between $\mathbf{WS}$ and $\mathbf{Y}$. In particular, a common algorithm for a regularized solution to the least squares problem is ridge regression. In this case, if $\beta \in \mathbb{R}^+$ is the L2 regularization factor chosen by model selection, the readout weights are computed in closed form as follows:

$$\mathbf{W} = \mathbf{Y}\mathbf{S}^T(\mathbf{S}\mathbf{S}^T + \beta\mathbf{I})^{-1} \tag{B.1}$$

Instead, the `IncFed` algorithm computes, for each client c, the matrices $\mathbf{A}_c \in \mathbb{R}^{N_y \times N_x}$ and $\mathbf{B}_c \in \mathbb{R}^{N_x \times N_x}$ as follow:

$$\mathbf{A}_c = \mathbf{Y}_c\mathbf{S}_c^T \quad \text{and} \quad \mathbf{B}_c = \mathbf{S}_c\mathbf{S}_c^T \tag{B.2}$$

The matrices $\mathbf{A}_c$ and $\mathbf{B}_c$ are sent to the server where they get summed as in the following equations:

$$\mathbf{A} = \sum_{c \in \mathcal{C}} \mathbf{A}_c \quad \text{and} \quad \mathbf{B} = \sum_{c \in \mathcal{C}} \mathbf{B}_c \tag{B.3}$$

After the summed matrices are computed, the server can compute the optimal readout weights $\mathbf{W}$ in closed-form as follows:

$$\mathbf{W} = \mathbf{A}(\mathbf{B} + \beta\mathbf{I})^{-1} \tag{B.4}$$

The matrices B.1 and B.4 are mathematically equivalent.

In `IncFed`, the number of floating-point values that get transferred from each client to the server is $N_x^2 + N_y N_x$.

B.2 Importance vector

The Fisher information matrix $\mathcal{F}$ was used successfully for parameter pruning in neural networks [28]. This gives us a straightforward way to save bandwidth, evaluating the importance measure of the parameters $\mathbf{W}$.

Let's consider a single instance $(\mathbf{S}_i, \mathbf{Y}_i)$ of states $\mathbf{S}$ and targets $\mathbf{Y}$. The linear model is $f(\mathbf{W}) = \mathbf{W}\mathbf{S}_i + \epsilon$, where ϵ is an error sampled by a normal distribution. Assuming the variance σ^2 is known, the log-likelihood of the data is given by the density:

$$\mathcal{L}(\mathbf{W}) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{-(\mathbf{Y}_i - \mathbf{W}\mathbf{S}_i)^2}{2\sigma^2} \tag{B.5}$$

The Fisher information matrix is the expected value of the negative of the Hessian matrix of $\mathcal{L}(\mathbf{W})$. So, we compute the gradient:

$$\mathcal{J}(\mathbf{W}) = \nabla_{\mathbf{W}} \frac{-(\mathbf{Y}_i - \mathbf{W}\mathbf{S}_i)^2}{2\sigma^2} = \frac{\mathbf{Y}_i \mathbf{S}_i^T}{\sigma^2} - \frac{\mathbf{W}\mathbf{S}_i \mathbf{S}_i^T}{\sigma^2} = \frac{(\mathbf{Y}_i - \mathbf{W}\mathbf{S}_i)\mathbf{S}_i^T}{\sigma^2} \tag{B.6}$$

Now, we compute the hessian:

$$\mathcal{H}(\mathbf{W}) = \frac{\partial}{\partial \mathbf{W}} \frac{(\mathbf{Y}_i - \mathbf{W}\mathbf{S}_i)\mathbf{S}_i^T}{\sigma^2} = \frac{\partial \mathbf{Y}_i \mathbf{S}_i^T}{\partial \mathbf{W}} - \frac{\partial \mathbf{W}\mathbf{S}_i \mathbf{S}_i^T}{\partial \mathbf{W}} = -\frac{\mathbf{S}_i \mathbf{S}_i^T}{\sigma^2} \tag{B.7}$$

so, the Fisher information matrix is:

$$\mathcal{F}(\mathbf{W}) = -\mathbb{E}\left[\mathcal{H}(\mathbf{W})\right] = \frac{\mathbf{S}_i \mathbf{S}_i^T}{\sigma^2} \tag{B.8}$$

Because gradients and Hessians are additive, if n data items are observed, the individual Fisher information matrices are simply summed:

$$\mathcal{F}(\mathbf{W}) = \frac{\sum_i \mathbf{S}_i \mathbf{S}_i^T}{\sigma^2} \tag{B.9}$$

and, if $\mathbf{S}^T = (\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n)$, $\mathcal{F}$ can be written as:

$$\mathcal{F}(\mathbf{W}) = \frac{\mathbf{S}\mathbf{S}^T}{\sigma^2} \tag{B.10}$$

The Fisher information matrix is used in classification problems with gradient optimization methods with probability density functions.

Instead, our problem is least squares solved by ridge regression and we need to translate the concept of Fisher information matrix. Since we are not dealing with density functions and that in the federated scenario we are interested in ranking the neurons, we can ignore the σ^2 value. The Fisher information matrix is reformulated as:

$$\mathcal{F}(\mathbf{W}) = \mathbf{S}\mathbf{S}^T \in \mathbb{R}^{N_x \times N_x} \tag{B.11}$$

The importance vector, used for the ranking of the neurons, is the sum over one of the axis of $\mathcal{F}$, and it is computed as:

$$\mathcal{I}(\mathbf{W}) = \mathcal{F}(\mathbf{W})\mathbf{1}_{N_x} = \mathbf{S}\mathbf{S}^T\mathbf{1}_{N_x} \in \mathbb{R}^{N_x} \tag{B.12}$$

where $\mathbf{1}_{N_x}^T = (1, 1, \ldots, 1) \in \mathbb{R}^{N_x}$ is the 1-vector of size N_x, and $\mathcal{I}$ represents how much each neuron is correlated with all the others neurons.

Also, if we consider $\mathcal{S}$ as the set of the top k neurons, we define the indicator function $\mathbb{1}_{\mathcal{S}} : \mathbb{R} \to \{0, 1\}$ that takes in input the importance, and returns in output the masked value. Such function is defined as:

$$\mathbb{1}_{\mathcal{S}}(x) = \begin{cases} 1 & \text{if } x \in \mathcal{S} \\ 0 & \text{otherwise} \end{cases} \tag{B.13}$$

If we consider a vector $\mathbf{X} \in \mathbb{R}^{N_x}$, with $\mathbb{1}_{\mathcal{S}}(\mathbf{X})$ means the element-wise application of the function $\mathbb{1}_{\mathcal{S}}(\cdot)$.

B.3 Partial Incremental Federated Learning

Partial Incremental Federated Learning (`Partial IncFed`) is an algorithm that uses part of the $\mathbf{A}_c$ and $\mathbf{B}_c$ matrices computed by `IncFed`, reducing the amount of data transfers.

The `Partial IncFed` algorithm computes, for each client c, the matrices $\mathbf{A}_c$ and $\mathbf{B}_c$ as in Eq. B.2 of `IncFed`. We notice that $\mathbf{B}_c$ is the hessian, so we consider it as the Fisher information as in Eq. B.11.

$$\mathcal{F}_c(\mathbf{W}_c) = \mathbf{S}_c \mathbf{S}_c^T = \mathbf{B}_c \tag{B.14}$$

In order to get the parameters importance, we compute for each client the importance vector as in Eq. B.12:

$$\mathcal{I}_c(\mathbf{W}_c) = \mathcal{F}_c(\mathbf{W}_c)\mathbf{1}_{N_x} = \mathbf{B}_c \mathbf{1}_{N_x} \tag{B.15}$$

At this point, if $\mathcal{S}_c$ is the set of the most important k neurons, each client c computes $\mathbf{A}_c^*$ and $\mathbf{B}_c^*$ as follows:

$$\mathbf{A}_c^* = \mathbf{A}_c \times \mathbb{1}_{\mathcal{S}_c} \quad \text{and} \quad \mathbf{B}_c^* = \mathbf{B}_c \times [\mathbf{I}_{N_x} \times (\mathbf{1}_{N_x} \otimes \mathbb{1}_{\mathcal{S}_c})] \tag{B.16}$$

where $\mathbb{1}_{\mathcal{S}_c}$ is used instead of $\mathbb{1}_{\mathcal{S}_c}(\mathcal{I}_c(\mathbf{W}_c))$ for the sake of conciseness, $\mathbf{I}_{N_x}$ is the identity matrix with size N_x and $\otimes$ is the outer product.

The matrices $\mathbf{A}_c^*$ and $\mathbf{B}_c^*$ are sparse with a certain number of non-zero elements and the rest are 0s. The indices and values of non-zero elements are sent to the server, where they are summed as sparse matrices like in the following equations:

$$\mathbf{A} = \sum_{c \in \mathcal{C}} \mathbf{A}_c^* \quad \text{and} \quad \mathbf{B} = \sum_{c \in \mathcal{C}} \mathbf{B}_c^* \tag{B.17}$$

The summed matrices are used by the server to compute the optimal readout weights $\mathbf{W}$ in closed-form as in Eq. B.4.

In `Partial IncFed`, the number of floating-point values that get transferred from each client to the server is $k(1 + N_x)$. If we consider $k = N_x/2$, `Partial IncFed` reduces the client floating-point values transmission by $\sim 60\%$ compared to `IncFed`.

B.4 Experiments

The aim of the experiments, is to measure and compare the performances of three different federation strategies: `IncFed`, `Partial IncFed` and `Random`. The `Random` strategy consists of selecting k random neurons indices and apply Eq. B.16. In the `Partial IncFed` we use a parameter $\alpha \in [0, 1]$ in order to consider $\alpha \times k$ important neurons, and $(1 - \alpha) \times k$ random selected neurons for exploration purposes.

For our experiments we used the WESAD dataset with 8 features and 4 classes as used in Bacciu et al. [2].

We use four degrees of availability of clients during federated learning: 25% (3 clients), 50% (5 clients), 75% (7 clients) and 100% (9 clients) of training subjects $\mathcal{C}$. The number of clients corresponds to the number of training subjects and the validation (3 clients) and test (3 clients) sets remains fixed in all experiments. The number of neurons to be considered are $k = (150/\left|\mathcal{C}\right|/100) \times N_x$. The best-performing hyper-parameters, which are selected by evaluating on the validation set, are shared across all models in the federation. Moreover, the models also share the same $\mathbf{W}_{in}$ and $\widehat{\mathbf{W}}$ matrices.

N_x	α
[100, 250, 500]	[0.3, 0.6, 0.8, 1]

Table B.1: Hyper-parameter ranges explored for the WESAD dataset.

	`IncFed`	`Partial IncFed`
N_x	500	500
α	-	0.3
λ	1e-6	1e-6
Leaking rate	1.0	1.0
Input scaling	0.9	0.9
Spectral radius	0.99	0.99

Table B.2: Models parameters.

Training users	`IncFed`	`Partial IncFed`	`Random`
100%	0.7801±0.77	**0.8128±0.70**	0.8084±0.74
75%	0.7504±0.81	**0.8108±0.76**	0.7887±0.86
50%	0.7223±0.80	**0.7642±0.93**	0.7600±0.95
25%	0.7093±0.89	**0.7565±0.96**	0.7480±1.01

Table B.3: Results of the experiments on WESAD. The values reported are the test accuracy and the standard deviation of the errors.

The preliminary results in Table B.3 show us how `Partial IncFed` produces regularization effects that lead to a better generalization, obtaining a test accuracy greater than the one of `IncFed` and keeping, at the same time, a significative reduction of the cost of communication among clients and server.

Bibliography

[1] Cisco annual internet report 2018–2023. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[2] Davide Bacciu, Daniele Di Sarli, Pouria Faraji, Claudio Gallicchio, and Alessio Micheli. Federated reservoir computing neural networks. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2021.

[3] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE internet of things journal*, 4(5):1125–1142, 2017.

[4] Mehdi Mohammadi and Ala Al-Fuqaha. Enabling cognitive smart cities using big data and machine learning: Approaches and challenges. *IEEE Communications Magazine*, 56(2):94–101, 2018.

[5] Yaohua Sun, Mugen Peng, Yangcheng Zhou, Yuzhe Huang, and Shiwen Mao. Application of machine learning in wireless networks: Key techniques and open issues. *IEEE Communications Surveys & Tutorials*, 21(4):3072–3108, 2019.

[6] Cisco global cloud index 2016-2021. https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/index.html.

[7] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. Security for the internet of things: a survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials*, 17(3):1294–1312, 2015.

[8] Teaching unipi project. https://www.unipi.it/index.php/risultati-e-prodotti/item/17229-teaching.

[9] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[10] Kenji Doya et al. Bifurcations in the learning of recurrent neural networks 3. *learning (RTRL)*, 3:17, 1992.

[11] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[13] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[14] Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34):13, 2001.

[15] Herbert Jaeger. Echo state network. *scholarpedia*, 2(9):2330, 2007.

[16] Jacob Poushter et al. Smartphone ownership and internet usage continues to climb in emerging economies. *Pew research center*, 22(1):1–44, 2016.

[17] Monica Anderson. Technology device ownership: 2015. 2015.

[18] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[19] Dinh C Nguyen, Ming Ding, Pubudu N Pathirana, Aruna Seneviratne, Jun Li, and H Vincent Poor. Federated learning for internet of things: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 23(3):1622–1658, 2021.

[20] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[21] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[22] Luc Pronzato and Andrej Pázman. Design of experiments in nonlinear models. *Lecture notes in statistics*, 212:1, 2013.

[23] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Outlier detection: A survey. *ACM Computing Surveys*, 14:15, 2007.

[24] Charu C Aggarwal. An introduction to outlier analysis. In *Outlier analysis*, pages 1–34. Springer, 2017.

[25] Raghavendra Chalapathy and Sanjay Chawla. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407*, 2019.

[26] Philip Schmidt, Attila Reiss, Robert Duerichen, Claus Marberger, and Kristof Van Laerhoven. Introducing wesad, a multimodal dataset for wearable stress and affect detection. In *Proceedings of the 20th ACM international conference on multimodal interaction*, pages 400–408, 2018.

[27] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.

[28] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.