University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Poisoning Attack Behavior Detection in Federated Learning

Timothy-Till Näscher
Zurich, Switzerland
Student ID: 18-935-338

Supervisor: Chao Feng, Dr. Alberto Huertas Celdran
Date of Submission: May 28, 2023

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract Deutsch

Federated Learning hat sich in Anwendungen, in denen die Daten der Nutzer hochsensibel sind als praktikable Alternative zu traditionellem Machine Learning erwiesen - besonders wenn die Endgeräte in der Lage sind, lokale Machine Learning Modelle selbst zu berechnen. Der Grossteil der jüngsten Forschungsarbeit auf dem Gebiet des Federated Learning ist auf Zentralisiertes Federated Learning ausgerichtet, bei dem lokal berechnete Updates von einem zentralen Server aggregiert werden um ein neues globales Modell zu erstellen. In letzter Zeit ist ein neuer Ansatz aufgetaucht: ein dezentralisiertes Netzwerk, in dem die Knoten selbst für die Aggregation von Modellupdates benachbarter Knoten verantwortlich sind. Das jüngste Interesse an diesem Ansatz hat das Interesse an der Schaffung eines Frameworks geweckt, das in der Lage ist, sowohl zentralisiertes als auch dezentralisiertes föderales Lernen zu simulieren. Idealerweise ist dieser Rahmen auch in der Lage, eine byzantinische Umgebung zu simulieren, in der Endgeräte böswillig sein können und versuchen, das globale Modell zu schädigen, indem sie falsche Daten oder verfälschte Modellaktualisierungen liefern. Dies würde den direkten Vergleich zwischen der zentralisierten und dezentralisierten Herangehensweise in diversen Szenarien, wie z.B. unterschiedlichen Angriffen oder Aggregationsregeln ermöglichen. Ein solcher Vergleich könnte wertvolle Erkenntnise zu deren Verhalten liefern und Vor-/Nachteile der beiden Herangehensweisen erleuchten.

Das FedStellar-Framework erfüllt diese Anforderungen, indem es die Möglichkeit bietet, beide Umgebungen in gutartigen Umfeld zu simulieren, jedoch werden bisher keine byzantinischen Szenarien unterstützt. In dieser Arbeit wird das FedStellar-Framework um zusätzliche Funktionen erweitert, indem weitere daten- und modellbasierte Angriffe und Aggregationsregeln implementiert werden. Zusätzlich werden die neuen Funktionen benutzt, um verschiedene Szenarien zu simulieren und untersuchen.

Abstract English

Federated Learning has emerged as a viable alternative to traditional Machine Learning in scenarios, where client data is highly sensitive and the client devices are capable of local computation of model updates. Much of the recent work in the area of Federated Learning was targeted towards the centralized setting, where locally computed model updates are aggregated by a centralized server to create a global model. Recently, a new approach has emerged: a fully decentralized network where clients themselves are responsible for aggregating updates of neighboring nodes. Recent interest in this setting has spurred interest in the creation of a unified framework, that is capable of simulating both centralized, as well as decentralized Federated Learning. Ideally, this framework is also capable of simulation under a byzantine setting, where client devices may be malicious and attempt to harm the shared model by providing false data or poisoned model updates. This would allow for direct comparison of the centralized and decentralized approach under various scenarios, such as different attacks or aggregation rules. Such a comparison could deliver valuable insights on their behavioral differences and benefits/disadvantages of either setting.

The FedStellar framework fills these requirements by providing capabilities to simulate either scenario with benign clients, yet so far it offers no support for the byzantine setting. In this thesis, the FedStellar framework is expanded with additional functionality, implementing more data- as well as model-based poisoning attacks and Aggregation Rules. Additionally, the new functionality is used to simulate and analyze different scenarios.

Acknowledgments

I would like to thank my supervisors Dr. Alberto Huertas Celdran and especially Chao Feng for his continual support and expert guidance throughout the duration of this thesis. I am equally appreciative of Prof. Dr. Burkhard Stiller for allowing me to complete my thesis in the Communication Systems Group.

Contents

Chapter 1

Introduction

In the last decade, Machine Learning (ML) has taken the world by storm. The ability to ”make sense” of enormous amounts of data and to gather valuable insights, without the need for manual engineering, has allowed ML models to be used anywhere, where large amounts of data are available. With the rise of the Internet of Things (IoT), as well as almost every person possessing a smartphone, the amount of data available is almost inconceivable. However, not all of this data can be freely shared with a central server for further processing and analysis, as it may contain personal, sensitive information. Not to mention, many of these edge devices are, in theory, perfectly capable of performing local model training themselves [1].

This practice of training an ML model across various, decentralized edge devices is known as Federated Learning (FL). Edge devices locally keep a ML model, continuing to train and improve the model using local data. These improvements are shared with a global model in the form of model updates, which are usually aggregated on a central server before being redistributed to the edge devices [1]. Most research pertaining to FL has been done under this Centralized Federated Learning (CFL) setting [2].

1.1 Motivation

Nowadays, edge devices are powerful enough to perform model training on their own. Therefore, it is not necessary to aggregate and redistribute the updated model from a central location. Instead, devices could share their updated model only with edge devices that are e.g in close proximity. This approach is called Decentralized Federated Learning (DFL). Some of the challenges of CFL, such as network and computational bottlenecks or trust issues, can be circumvented by using the DFL approach. For now, only limited academic work has been done on DFL, especially on Poisoning Attacks in DFL settings. Most papers implementing some kind of Attack or Defense method for FL only test on the centralized setup. In contrast, the DFL setting has received little attention so far. Additionally, there is little research comparing the performance of the CFL setup against the DFL setup [2].

This gap in research may also be attributed to the fact, that the lack of a unified framework capable of simulating both the CFL as well as DFL scenario makes efficient comparison between the settings difficult. The FedStellar framework fills this gap by providing a powerful and comprehensive framework capable of simulating both scenarios with different model architectures and datasets. Yet the lack of support for the byzantine setting, where clients may be malicious and send poisoned data or updates, makes comparisons on the robustness of a given scenario difficult. The goal of this thesis is to expand this framework with various data-based and model-based poisoning attacks, as well as additional Aggregation Rules. The framework will then be used to analyze the behavioral differences between CFL and DFL.

1.2 Description of Work

FedStellar is a framework that can be used to configure, deploy and evaluate Federated Learning scenarios. At the time of starting this thesis, FedStellar did not offer any support for adversarial scenarios. Thus, no aggregation rules apart from FedAvg were available. In this work, the FedStellar Framework was extended to support adversarial scenarios as well as more complex, byzantine-robust aggregation rules. Together with the attacks and aggregation rules added by my supervisor, FedStellar now supports a comprehensive suite of data manipulation and update manipulation attacks as well as the most relevant aggregation rules. In its current state, the FedStellar Framework can be used to evaluate any combination of Federation (CFL, DFL or SDFL), adversarial context and aggregation rule - all in a user friendly manner using the web interface. Another part of this work is the comparison of centralized and decentralized setups, especially under attack. Emphasis is also given to the comparison of the behaviour of different decentralized network topologies (such as Ring or Fully connected).

1.3 Thesis Outline

The second chapter covers the background of this thesis. The reader is introduced to the challenges of FL in general, as well es the different types of FL. Some applications of FL are covered as well. In section 2.3, the framework ”FedStellar” is presented. The third chapter gives an overview of related work, where different attacks and defense methods (aggregation rules) are explained. The following section 4 shows the newly implemented attacks and defense methods in FedStellar. They are evaluated and compared in section 5. Chapter 6 summarizes the work and draws conclusions from the evaluations.

Chapter 2

Background

In this chapter, an overview of Machine- and Federated Learning is given. First, some information is provided on the traditional Machine Learning setting, with short introductions to some important data preprocessing steps, as well as a broad overview of the training procedure. Next, the relevant ML architectures used in the FedStellar framework are introduced. A list of challenges that arise in the traditional ML setting is also given.

Following this, a similar introduction is given on the Federated Learning approach. An overview of the different types of federations are given, how they are generally trained and how their approaches differ. Some different network topologies for the decentralized scenario are also discussed. After some different application scenarios of Federated Learning are presented, the chapter is concluded by briefly talking about the challenges that arise in the Federated Learning setting.

2.1 Machine Learning

Few advances have taken the world of technology by storm as much as Machine Learning (ML) has in recent years. It is without a doubt one of the most powerful tools available to make sense of and utilize large amounts of data. Especially in the present age of information, where data is ubiquitous and advances in computational power allow for the creation of powerful models.
However, it did not start out in such a grandiose fashion. In this section, a brief overview is given on ML and the model architectures that were used in the practical part of this thesis.

2.1.1 Supervised and Unsupervised Learning

Machine Learning can take many forms, but it is generally categorized into two types, supervised and unsupervised learning. The difference is that supervised ML requires labeled training data, where the objective of the training is to learn the underlying distribution

of the given training data, in order to predict or classify new samples that are similar to the training data distribution [3].
Unsupervised learning on the other hand does not require labelled training data. The objective of the training is to learn trends and relationships between samples of the training dataset, often used for exploratory data analysis (EDA) [3].

2.1.2 Data Preprocessing

An important first step when doing ML is data preprocessing. As the saying goes, "garbage in, garbage out". Two important concepts for general preprocessing of data will be introduced in this subsection.

Normalization

A key concept of data preprocessing is *normalization*. As an example, when working with RGB images, each color channel is generally discretized using 8 bits, leading to values in the range [0,255]. This can be problematic during training, as dark colors (high values) will produce vastly different outputs than bright colors (low values). A solution is to *normalize* inputs to a certain range. More specifically, the goal is to achieve a mean value of 0 across all pixels while changing the standard deviation to be 1. More formally, given an input image $\mathbf{x}$, the normalized input $\hat{\mathbf{x}}$ is given by:

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - E[\mathbf{x}]}{\sqrt{\text{Var}(\mathbf{x})}}. \tag{2.1}$$

Images can be normalized individually (known as instance normalization) or across a batch of images (known as batch normalization[4]) [5].

One-Hot Encoding

Next to scalar data which may be normalized, some different considerations need to be taken for categorical data. An example of categorical data could be a set of colors, e.g $(red, blue, green)$. Neural Networks however require some numerical representation of data. A naive approach could be indexing them with the numbers $(0, 1, 2)$. This would however imply some underlying ordering between colors, where red would be more similar to blue than green and misclassifying a green object as red would be penalized more heavily than a misclassification of a blue object. An intuitive way to alleviate this problem is using *one-hot encoding*. Given categorical data with n categories, indexed by $i = \{1, 2, ..., n\}$ the i-th category would be encoded as an n-dimensional vector of zeros, where only the i-th dimension contains a 1. In the color example, this would map the set $(red, blue, green)$ to the one-hot encoding $([1\ 0\ 0]^T, [0\ 1\ 0]^T, [0\ 0\ 1]^T)$ [6].

2.1.3 Training Procedure

During training, the goal is to iteratively update the values of the parameters inside the neural network. The objective of training is to minimize a given objective function, often known as a *loss function* $\mathcal{L}(\cdot;\theta)$ parameterized by the network parameters θ. The goal of the loss function is to compare the output of a neural network with the correct output as specified by the label of the data. Similar outputs yield low values, while large differences are penalized. The loss function chosen heavily depends on the given task. For regression tasks, i.e predicting continuous values such as the temperature of tomorrow's weather, a popular loss function is Mean Squared Error (MSE) [7]. Given a target label y and a predicted value $\hat{y}$, the MSE loss is formulated as:

$$\mathcal{L}(\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\left(y[i] - \hat{y}[i]\right)^2, \tag{2.2}$$

On the other hand, classification tasks, i.e predicting a one-hot encoded categorical value, is often done using binary cross-entropy(BCE)[8]:

$$\mathcal{L}(\hat{y}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y[i]\log(\hat{y}[i]) + (1-y[i])\log(1-\hat{y}[i])\right], \tag{2.3}$$

The goal of training is to minimize the given loss function $\mathcal{L}$ with respect to our parameter vector θ. Formally, our optimization problem is:

$$\min_{\theta} \mathcal{L}(\theta) \tag{2.4}$$

Geometrically speaking, the gradient ∇_f of a function f can be viewed as a vector pointing towards the direction of steepest ascent, much like the one dimensional derivative depicts the slope at a given point of the function. One could iteratively optimize the model's parameters by incrementally moving down the gradient of the loss function, i.e

$$\theta \leftarrow \theta - \eta \cdot \Delta\theta. \tag{2.5}$$

This is known as gradient descent (GD). Due to the often very large amount of data available for training, it is ill-advised to compute the gradient using the entire training data. This approach would require a large amount of computation for each parameter update, while at the same time increasing the chance of landing in a local minimum. Instead, a popular approach is Stochastic Gradient Descent (SGD), only calculating the gradient on a batch of data (e.g 32 or 128 images) at a time. Another popular optimization algorithm is Adaptive Moment Estimation (Adam) [9], which is based on SGD but incorporates an adaptive learning approach, making it more robust for problems with sparse or noisy gradients [5], [3].

Application in a Production Setting

When providing a service that involves ML, the model is often highly complex and requires a large amount of storage to save the model parameters. Edge devices often do not have

the required amount of storage and computational power to keep a local model, let alone run it locally. Therefore the model is often kept on a central server with edge devices requiring an active internet connection in order to make use of this global model. Figure 2.1 shows the Pipeline of a "traditional" ML Application, where the edge devices share their data with the server in order to train a global model. This global model is redistributed to the edge devices and no updates to the model are applied by the clients [10].

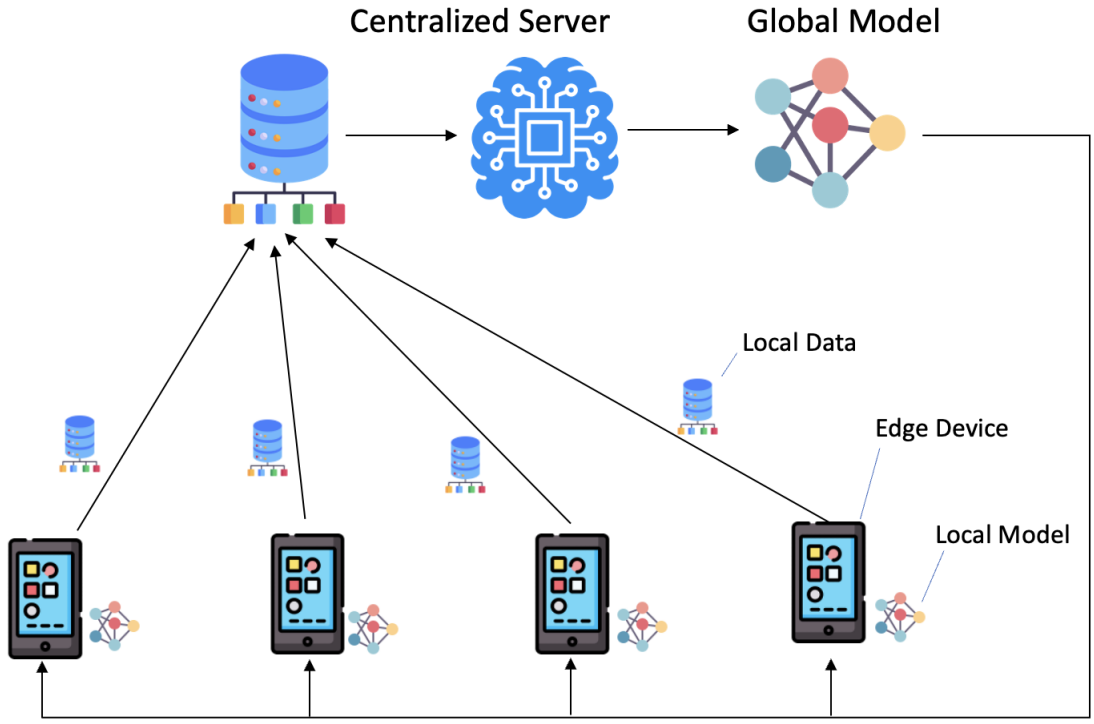


Figure 2.1: Machine Learning Pipeline

2.1.4 Multilayer Perceptron

One of the earliest ML architectures that is still in use today is the Multilayer Perceptron (MLP). Its architecture is inspired by the functioning of the brain, where a large number of neurons are connected to each other and form pathways throughout the brain, with electrical impulses traveling along them. When a neuron is stimulated by an electrical signal, it may choose to pass the impulse on towards connected neurons or block the signal from traveling further [11].
In the case of the MLP, neurons are grouped into layers. The first layer is called the input layer. Subsequent layers are called hidden layers, terminated by the final layer, called output layer. Each neuron in a layer is connected to all the neurons of the following layer, with each connection being given a weight to indicate the amount of information it passes to the given neuron in the following layer [11]. Information flows from the input layer through the hidden layers of the network, with final predictions being provided by the

output layer. Due to this flow of data, they are a type of feedforward neural network (FNN) [3].

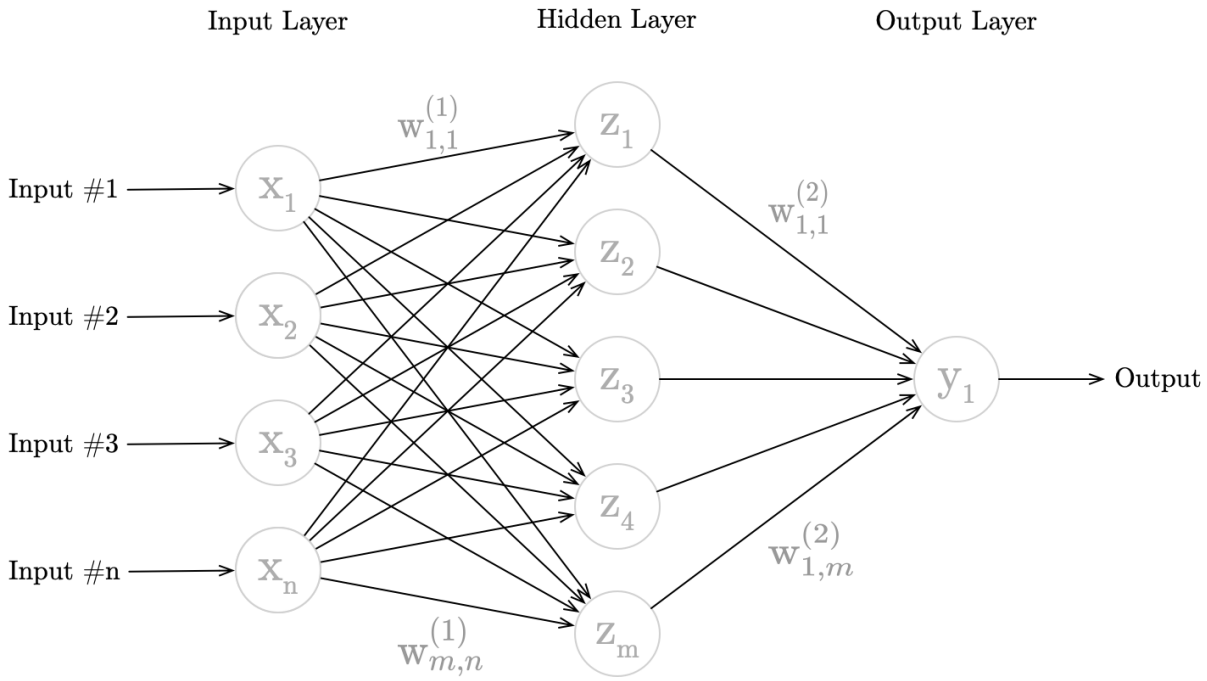


Figure 2.2: Example of an MLP architecture

Given a network with input neurons $\mathbf{x} \in \mathbb{R}^n$ and a single hidden layer with neurons $\mathbf{z} \in \mathbb{R}^m$, where the weight of the connection from the i-th neuron of the input layer to the j-th neuron of the hidden layer is given by $\mathbf{w}^{(1)}_{j,i} \in \mathbb{R}^{n \times m}$, the value of the k-th neuron in the hidden layer is given by:

$$z_k = \sigma\left(\sum_{l=1}^{n} w^{(1)}_{k,l} x_l + b^{(1)}_k\right), \tag{2.6}$$

where σ is called the *activation function* and $\mathbf{b} \in \mathbb{R}^m$ is called *bias*. The activation function ensures that the output is scaled to a known interval, in order to prevent the gradient from vanishing or exploding during backpropagation. Additionally, activation functions are generally non-linear, allowing the network to learn non-linear mappings. The bias on the other hand is used to shift the output of the activation function. Finally, the output y can be calculated in a similar fashion, i.e:

$$y = \sigma\left(\sum_{l=1}^{m} w^{(2)}_{1,l} z_l + b^{(2)}\right), \tag{2.7}$$

where the activation function of the final layer depends on the task. In case of a regression task it may be omitted, while a softmax activation is generally used for classification tasks to produce a predicted probability for each class [12].

2.1.5 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are another very popular type of feedforward neural network, which finds many of its uses in the field of Computer Vision (CV). Its power lies in its abilities to extract useful information from grid-like structures [13], making it ideal for feature extraction on images. Just as with MLPs, CNNs also consist of multiple layers where information flows from the input layers through the hidden layers to the output layers. The key difference lies in the types of layers that are used. MLPs use fully connected layers throughout their structure, while CNNs consist of multiple convolution and pooling layers, terminated by a fully connected layer at the end to yield the final predictions.

While the first CNN architecture was proposed all the way back in the 1980s [14], they only started taking off in the early 2000s with technological advances in computational power, namely GPU-based computing.

Convolution Layers

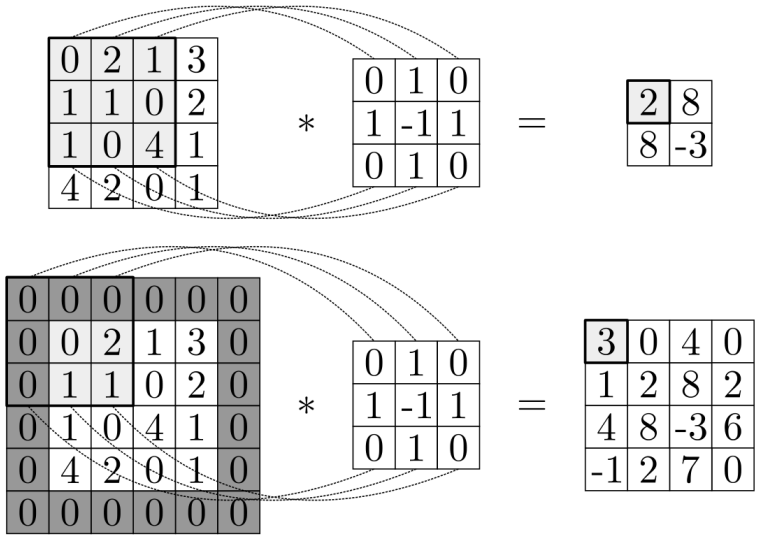


Figure 2.3: 2D Convolution Operation
Shown without (top) and with (bottom) zero padding [13].

The core concept that CNNs make use of, is, as the name implies, the so called *convolution operator*. The convolution operator in the discrete 2D scenario is generally a square matrix which slides over the input image, extracting various information such as frequencies or derivatives (gradients) along the way. The output of convolving an image with a convolution operator (also called kernel in this setting) is known as a feature map. It is calculated by the summation of element-wise multiplications of the kernel and the input feature map [13]. An illustration of a convolution operation can be seen in Figure 2.3.

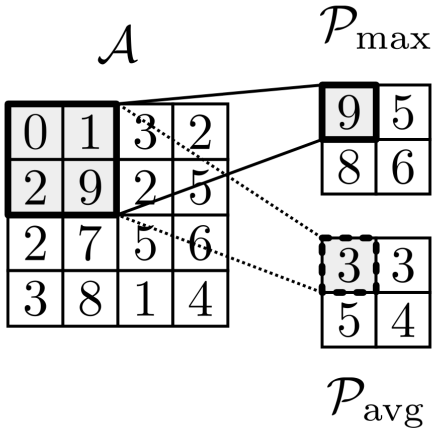


Figure 2.4: Max-Pooling and Average-Pooling [13]

Pooling

Sometimes it is desirable to intermittently reduce the size of the feature map. This has various benefits, such as helping with generalization due to the aggregation of information, providing additional translation invariance to the network while also reducing the memory footprint of the network. Similarly to the kernel in convolutional layers, pooling is done by sliding a window over the feature map and aggregating the information inside the window.The two main pooling operations are max-pooling and average-pooling. While max-pooling aggregates by taking the largest value inside the window, average-pooling returns an average of all the values inside the window, as seen in figure 2.4 [13].

2.1.6 Challenges of Machine Learning

With all its benefits, there are also some difficulties that need to be overcome in order to apply ML to a production setting. These problems range from difficulties in sourcing sufficient training data to data security and also lack of interpretability when predictions do not match expectations. In this subsection, a broad overview of challenges in ML is given.

Data Collection and Preprocessing

The first step towards any kind of ML is gathering enough training data. Depending on the complexity of the problem, millions of training samples need to be provided in order for the model to be able to properly deal with new, unseen data. Due to this need for enough data, a lot of the data being collected by organizations is not relevant to the problem a given ML model is required to solve, which is why the data also needs to be filtered and preprocessed for relevant information. In the supervised setting, not only are these enormous amounts of data required, the data also needs to be labelled, both labour- and time-intensive processes [15].

Privacy and Security Concerns

Growing privacy concerns prove to be an additional hurdle for the collection of training data. Collected data may contain sensitive information, requiring additional considerations in regards to data storage and encryption. Malicious users could additionally provide intentionally wrong information, a large enough amount of which could lead to difficulties during training as well as degradation in model performance [16], [17].

Difficulties during Training

Even with approriate data available, the training procedure itself often also proves to be difficult. Training a model for too long of a time, overparameterizing or using too many features during training may all lead to overfitting, causing a loss of generalization as the model learns the trained data too well and is no longer able to deal with unseen data. The same goes for the other direction as well, training for too little time, using a model that is not complex enough or using too little features to properly represent the problem leads to underfitting, where the model does not learn the training data well enough in order to predict on new examples [18]. Since ML models are essentially a blackbox, it often proves difficult to interpret the results when a model does not behave in its intended way.

2.2 Federated Learning

Federated Learning (FL) is a relatively new kind of Machine Learning , which trains a global model using data gathered from multiple decentralized edge devices that do not share information. This approach offers multiple benefits, due to decentralizing both the data used for training, as well as the computing power available. It was first introduced by Google in 2017, where they used FL to improve on the suggestion query model of Gboard (a virtual keyboard app). Since then, it has been applied to a wide number of different industries due to the benefits it offers [1].

Unlike in traditional ML where the model is trained on a singular device, in FL each edge device individually computes updates for the shared model. As the training data only needs to be available locally, the updates that are sent to the shared model do not compromise on the privacy of the data [19].

2.2.1 Centralized Federated Learning

In its original paper, [1] coined the term *Federated Learning*, "since the learning task is solved by a loose federation of participating devices (which we refer to as clients) which are coordinated by a central server". In the centralized FL approach, the global model is first trained on a central server using already existing data. A snapshot of this global model is subsequently shared with all the clients. Following this, every client uses the received model in their individual way, collecting additional training data along the way. This

data is used to improve the local model further. It is important to note that the collected data is only processed locally, in order to preserve the privacy of clients. Periodically, the data gets sent to the central server in the form of model updates (either by sending parameter gradients or the parameters themselves). A general outline of a centralized learning process is given as seen in [20]:

1. **Client Selection:** As the number of clients may be in the millions, it is not feasible to update the global model with each locally computed model update (diminishing returns set in, see [1]). The central server selects a subset (for example randomly, see [21]) of available clients that meet some given eligibility requirements (in the case of mobile phones e.g plugged in, connected to WiFi, idle).

2. **Broadcast:** The selected subset of clients receives the current model weights and a training program from the central server.

3. **Client computation:** The selected clients perform a model update by executing the given training program on the local data (e.g Stochastic Gradient Descent (SGD)).

4. **Aggregation:** The computed model updates are subsequently sent to the central server, where the updates are first aggregated. This step allows the inclusion of various techniques to increase security (rejecting malicious updates), privacy (secure aggregation, noise addition) or efficiency (update compression).

5. **Model update:** The central server locally computes an update of the global model using the aggregated update that was computed and redistributes it to the clients.

An illustration of this update cycle is shown in Figure 2.5.

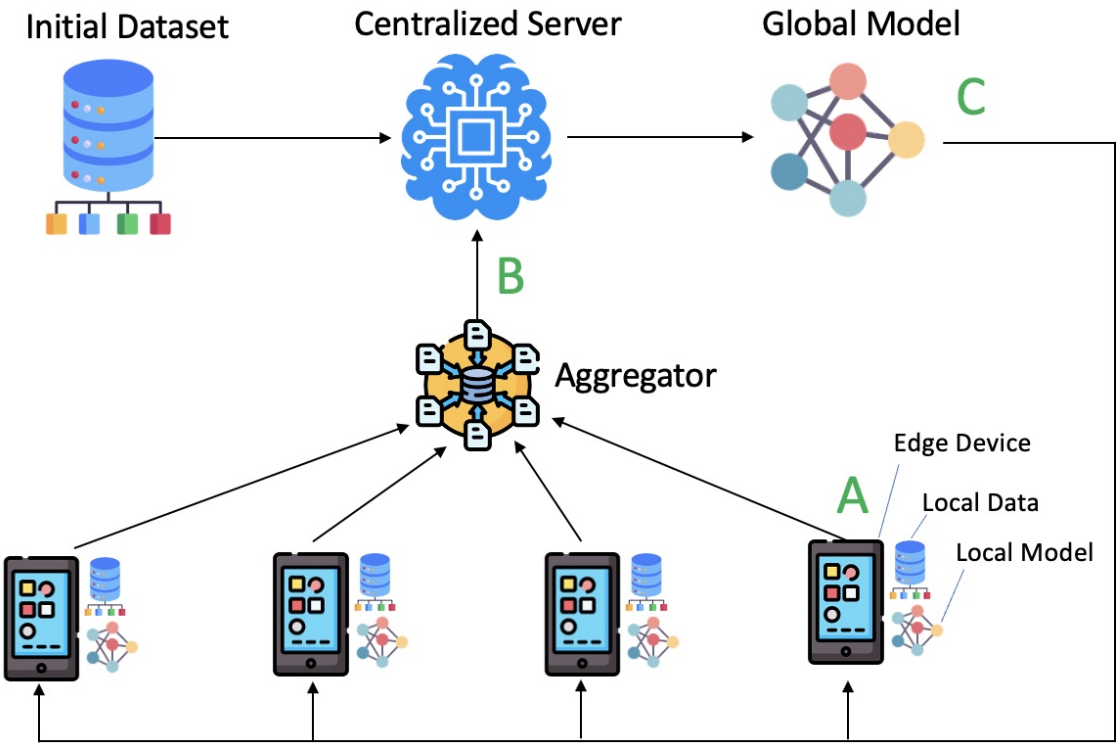


Figure 2.5: CFL Learning Process
During usage, the edge device locally processes and updates its current model (A). The locally collected data is periodically sent to a central server, where the data from various edge devices is first aggregated (B), after which it is used to update a shared model that gets sent back to the clients (C) [22]

2.2.2 Decentralized Federated Learning

In DFL, the clients communicate with each other on a network instead of relying on a central server. This communication network may be constructed in different ways, providing different communication routes and neighbors for each participant. The performance, efficiency and robustness of the entire DFL system depends on this topology [23]. Section 2.2.4 gives an overview about different network topologies.

A general outline of a centralized learning process is given as seen in [2]: The learning process in DFL can be split up into 4 steps: Local model training, parameter (updates) exchange, local model aggregation and parameter (updates) exchange again [2].

In a DFL setup, every client is assigned one (or more) roles, defining which task of the DFL learning process is done by them. There are four roles: *trainer*, *aggregator*, *proxy* and *idle*. A *trainer* trains a local model using its own dataset. After completing training, model parameters are sent to aggregators. The trainer node then receives parameters after aggregation to update his local model. The *aggregator* receives parameters from neighboring nodes which it then aggregates. The resulting parameters are sent back to the nodes. Direct communication between aggregators and trainers is not possible in some network topologies (for example in clustering topologies, see Figure 2.9). Then, a *proxy* node is needed to forward parameters - with that, communication between different network topologies is possible. Finally, it is also possible for a node to be *idle*. An idle node does not participate in the learning process of the network - no parameters are sent.

DFL architectures are categorized based on the type of federation, which depends on the types of nodes participating. Current literature differentiate two types: *cross-silo* and *cross-device*. In cross-silo DFL, the nodes are typically organizations or data centers. The entire setup has few participants but a large amount of data each. The nodes usually have high performance computing and reliable networking to their disposal. Cross-device DFL involves a larger number of nodes, typically edge devices or wearables, with each node having a smaller amount of data and limited computational power. The communication network is usually weaker than in cross-silo DFL [24], [2]. Depending on the distribution of data between the nodes, DFL architectures can be further sorted in three categories: *Horizontal Federated Learning* (HFL), *Vertical Federated Learning* (VFL) and *Federated Transfer Learning* (FTL). HFL refers to a scenario where the nodes share the same set of features but different samples. In VFL, the nodes share the same samples but have different features. FTL describes a scenario where the participants share neither the samples nor the feature space [25].

2.2.3 Semi-Decentralized Federated Learning

Semi-Decentralized Federated Learning (SDFL) differs from DFL in that the last two steps of the learning process, model aggregation and distribution are performed by a single node ("Leader"). The "leadership" is usually transferred from one node to another. The selection of the new "Leader" can be random, or based on performance metrics such as network or computational capacity [26]. As the "Leader"-Node represents a bottleneck

in the system, the choice of "Leader", i.e. the selection mechanism has an impact on the performance and robustness of the system [2].

2.2.4 Network Topologies

For a DFL scenario, the choice of network topology is very important as it impacts the robustness, performance and efficiency of the scenario [2]. In this section, the different network topologies are introduced and compared in greater detail.

Fully connected

In this network topology, all nodes have a direct connection to each other (see Figure 2.6. For every new node in the system, a link has to be added to each other node. With that, the expense and complexity of managing the connections of each node increase - the amount of links increases quadratically. However, the large amount of links allows the network to stay functional even when a few nodes or connections may fail. The fully connected topology is, generally speaking, the most robust and fault tolerant topology [2].

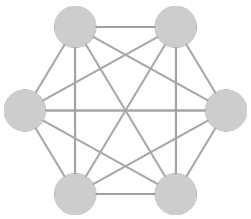


Figure 2.6: Fully Connected Topology

Ring

In a ring-topology network, the nodes are connected in a circle. Each node has two neighbors and a link to each of them. Regardless of the network size, every node only maintains two connections. Therefore, the communication costs only increase linearly. However, the delays in transmission increase with the network size. If each node sends his parameters for update to both neighbors, the network is called a *bidirectional* network. If each node only contacts one neighbor, the network is called a *unidirectional* network. Bidirectional networks are more reliable and fault tolerant than unidirectional networks. Generally, ring networks are more resilient in adversarial scenarios, but lack fault tolerance in comparison to the fully connected topology. [2].

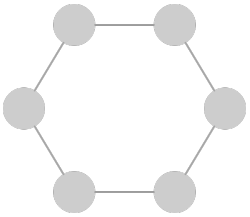


Figure 2.7: Ring Topology

Star

Star networks are constructed similarly to Centralized FL setups. One central node is acting as a proxy and manages all communication links with other nodes. Every new node only needs one link to the central node. Due to the central node being a single point of failure, the star network doesn't provide good fault tolerance. Also, the networks performance is limited by the central server, posing a bottleneck. Due to the small amount of communication links, the communication costs are low and scale linearly [2].

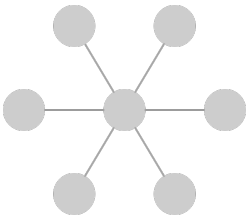


Figure 2.8: Star Topology

Node clustering

The node clustering topology groups the nodes into hierarchical clusters. For example, similarity-based clusters group the nodes depending on their local model parameters. After the grouping, the nodes in each cluster have similar models and a more even performance. The connected clusters can also be of different topology, for example a ring network connected to a fully connected network via a proxy. [2] In Figure 2.9 and 2.10 the red colored nodes act as a proxy. 2-Clique is a special case of node clustering - there are 2 node clusters, which are connected by a single link. The nodes in the two clusters are fully connected to each other [27].

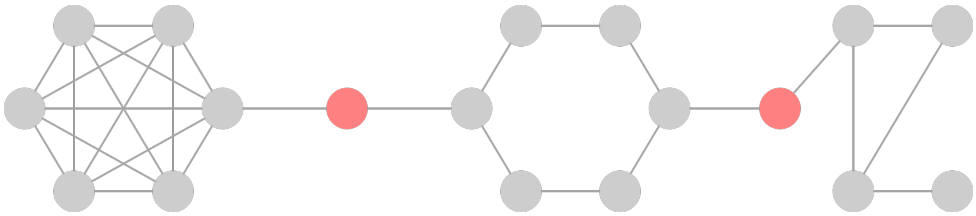


Figure 2.9: Node Clustering Topology

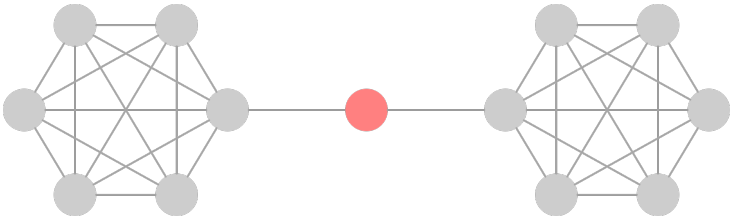


Figure 2.10: 2-Clique Topology

2.2.5 Applications of Federated Learning

Due to the flexibility of FL systems, they are able to be used in a wide variety of scenarios. A non-comprehensive overview of some of the possible applications is given in this section.

Healthcare

Due to the sensitive nature of private healthcare data, it does not come as a surprise that FL is applied in this sector. The exchange of information between institutes, hospitals and federal agencies is an unavoidable necessity. Federated Learning allows this exchange to happen in an encrypted manner, as each entity only needs to share model updates rather than health records. For instance, [28] proposes a community-based DFL algorithm for the prediction of mortality and hospital stay time. [29] proposes further improvements on communication efficiency for DFL over a graph, which was later demonstrated to be effective in the detection of Alzheimer's disease. More recently, FL was also used to help detect COVID-19 from computed tomography (CT) chest scans [30].

Industry 4.0

With the advent of Internet-of-Things (IoT) and cloud computing, FL has also found numerous applications in industrial settings. The authors of [31] talk about its uses in mission-critical Device-to-Device (D2D) operations, such as e.g collaborative robots (cobots), machines that are able to safely operate alongside humans in a shared workspace. Blockchain-enabled federated learning has also been used for cognitive computing, the idea of recreating the thought processes of human brains [32]. Finally, cybersecurity is also a big topic in industrial settings. [33] used peer-to-peer federated learning for anomaly detection, where training data may be locally re-balanced to improve performance in non-IID settings. A DFL framework with hierarchical blockchain architecture was also used to enhance privacy in the industrial metaverse [34].

Mobile Services

The first application of FL was tested on the next-word prediction of Google's on-screen keyboard app, Gboard [1]. Following its inception, it was further used in a wide variety of

everyday tasks. For example, Apple is using FL to improve on-device personalization (e.g news feed, automatic speech recognition (ASR))[35]. It also finds its uses in improving recommender systems [36], [37]. Next to the improved privacy of FL, a key benefit is the ability to utilize the combined processing power of available edge devices while at the same time reducing the distance between the processing unit and data storage. In this regard, numerous works have explored the possibilities of further reducing bandwidth and response times [38] [20].

2.2.6 Challenges of Federated Learning

With all its benefits, there are also some challenges associated with FL. Due to the intrinsic constraints of edge-devices as well as the data used in a FL setting, some considerations need to be taken into account that do not exist in the classical ML setting. Four main issues will be presented in the following:

System Heterogeneity

Unlike in traditional ML, where computational power is bottlenecked by the current state of technology, FL models are trained on often heavily resource-constrained edge devices. Not to mention that in many cases, the available resources also differ amongst clients. In the case of smartphones for example, there is large variability in processing power (CPU, memory), available communication bandwidth (3G, 4G, WiFi) or battery capacity available. Additional care must be taken to ensure that the models do not interfere with the daily usage of clients, regardless of their device type. This means, for example, that resource-heavy gradient updates may only happen at night, when the phone is not being used and it is connected to power. A similar issue is also prevalent with low-energy devices, such as microcontrollers, as they may only infrequently send gradient updates [39].

Statistical Heterogeneity

A major issue in FL is the distribution of data amongst clients. Since in the FL setting the kind of data that is collected strongly depends on the client's usage of the service, the data locally held by each client is non-representative of the population distribution and as such non-IID. Depending on the amount of usage from clients, the amount of data available on each device may also be vastly different, leading to unbalanced data [21]. As devices may only be active during a select time (e.g at night, while charging), the nature of collected data may also change depending on the current time. Clients residing in America likely transmit their data at a different time than clients that live in Asia[40] [41].

Privacy

The data that is held by clients is often sensitive and should not be seen by others. A big concern therefore is the encryption of data during the model update step. While it is not necessary to transmit raw data in the form of audio or images, in theory improving the general privacy of the system, it has been shown that it is still possible to extract identifiable data from gradient updates [42]. While encryption may increase the privacy in FL systems, it will put additional strain on the already resource-constrained edge-devices often used for FL. It is therefore important to balance computational availability with privacy concerns [21].

Bandwidth

Due to the need to share the locally calculated model updates with other participants or a centralized server, it is important for the updates to consider the amount of available bandwidth on the client's devices. They are usually heavily constrained, with communication being slow and often very expensive [43]. Additionally, the number of client devices inside a FL system is often in the millions, furthering the need for efficient and lightweight communication.

2.3 FedStellar

Fedstellar is a framework that can be used to configure, deploy and evaluate Federated Learning scenarios. It allows the user to create centralized, decentralized and semi-decentralized scenarios. It supports different datasets (MNIST, FEMNIST [Fashion-MNIST, CIFAR-10 added in this thesis]), different attacks on the scenarios (Untargeted Data Manipulation [Targeted Data Manipulation, Update Manipulation added in this thesis]), as well as different Aggregation methods (Krum, Median, FedAvg, TrimmedMean [Bulyan added in this thesis]). In decentralized scenarios, the topology of the network can be changed (the topologies mentioned in Section 2.1.3 are available and preconfigured). After a scenario has been run, many kind of statistics are available to monitor and analyze the performance and resource consumption. Statistical data is available for export for further analysis outside of FedStellar. The data is displayed in a TensorBoard instance [44]. A general overview of the architecture of FedStellar is shown in Figure 4.1.

2.4 Datasets

MNIST

The MNIST dataset contains grayscale images of handwritten digits (10 classes, labelled 0-9). It consists of 70'000 images in total, with 60'000 being used for training and 10'000 for testing. The images are grayscaled and have a resolution of 28×28 pixels (represented by a $1 \times 28 \times 28$ Tensor) [45].

EMNIST

The EMNIST (Extended MNIST) dataset contains grayscale images of handwritten digits and characters (62 classes, labelled 0-9, a-z, A-Z). It was derived from the same dataset (NIST Special Database 19) as MNIST, but also uses the characters. The full dataset consists of 814'255 images in total, with 697'932 used for training and 116'323 for testing. Other, smaller versions such as the "balanced EMNIST" are smaller, with 131'600 images in total, with 112'800 for training and 18'800 for testing. The images are grayscaled and have a resolution of 28×28 pixels (represented by a $1 \times 28 \times 28$ Tensor) [46].

FEMNIST

The FEMNIST (Federated Extended MNIST) dataset contains grayscale images of handwritten digits and characters (62 classes, labelled 0-9, a-z, A-Z). It was built specifically for federated setups by partitioning the EMNIST Dataset [46]) data based on the writer of the digit / character. It is supposed to cater to a Federated Learning setup, where every node gets the data of one writer (to simulate a user). It consists of 70'000 images in total, with 60'000 being used for training and 10'000 for testing. The images are grayscaled and have a resolution of 28×28 pixels (represented by a $1 \times 28 \times 28$ Tensor) [47].

CH-MNIST

The CH-MNIST (Colorectal Histology MNIST) dataset contains histological images of human colorectal cancer (8 different classes of tissues). It contains 5000 150×150px images that contain only one type of tissue as well as 10 larger 5000×5000px images that contain multiple tissue classes. The images are colored (represented by a $3 \times 150 \times 150$ resp. $3 \times 5000 \times 5000$ tensor) [48].

CIFAR-10

The CIFAR-10 dataset contains color images of 10 different classes (1: airplane, 2: automobile, 3: bird, 4: cat, 5: deer, 6: dog, 7: frog, 8: horse, 9: ship, 10: truck). It consists of 60'000 images in total, with 50'000 being used for training and 10'000 for testing (1'000 images per class).The images are colored and have a resolution of 32×32 pixels (represented by a $3 \times 32 \times 32$ Tensor) [49].

Fashion-MNIST

The Fashion-MNIST dataset contains grayscale images of clothing items from 10 different classes (1: t-shirt/top, 2: trouser, 3: pullover, 4: dress, 5: coat, 6: sandal, 7: shirt, 8: sneaker, 9: bag, 10: ankle boot). It consists of 70'000 images in total, with 60'000 being used for training and 10'000 for testing. The images are grayscaled and have a resolution of 28×28 pixels (represented by a $1 \times 28 \times 28$ Tensor) [50].

Chapter 3

Related Work

In this section, the reader is introduced to the most important papers on Federated Learning, posioning attacks as well as defense methods / aggregation rules.

In 2016, the concept of Federated Learning was introduced for the first time in [1]. They define federated learning as the concept of leaving the training data on mobile devices and learning a shared model on a central server by aggregating updates computed on the mobile devices which cannot share their data with the central server (for example due to privacy concerns) [1].

3.1 Attacks on Federated Learning

The following section will give an overview about the different categories of attacks. Table 3.3 gives an overview about literature that introduces attacks that belong to the categories listed below.

Data Poisoning Attacks can be clustered in two groups: *targeted* and *untargeted* attacks. Targeted attacks aim to maximize the decrease in global model accuracy on only one (or more) class(es). All samples not belonging to the targeted class(es) will be correctly classified. Untargeted attacks do not target a specific class, but try to reduce the accuracy on any class (the entire distribution). Data poisoning attacks in general do not disturb the update or training process - only the training data is being manipulated [51].

Model poisoning attacks manipulate the parameters (model) that is sent to the server for aggregation [52].

Backdoor attacks aim to make the global model misclassify on inputs containing a specific trigger pattern. If this pattern is available in the original distribution (for example red birds in CIFAR-10), the attack is referred to as a *semantic backdoor*. All samples not containing the pattern will be correctly classified. If the trigger is inserted into the original data (for example a white circle at a certain position) the attack is referred to as an *artificial backdoor*. All samples will be correctly classified except where the trigger was inserted [51].

3.1.1 Data Poisoning / Manipulation

[53] present a study on targeted poisoning attacks where malicious clients attack the global model by sending updates created by training the local model on mislabeled data ("label flipping"). This targeted attack changes one specific source class to another: ($c_{src} \rightarrow c_{tgt}$). For more details on the attack see Algorithm 1. They evaluate three settings for choosing the source label to attack, depending on their behaviour in federated non-poisoned training:
I: $c_{src} \rightarrow c_{tgt}$ where c_{src} was often misclassified as c_{tgt}
II: $c_{src} \rightarrow c_{tgt}$ where c_{src} was rarely misclassified as c_{tgt}
III: $c_{src} \rightarrow c_{tgt}$ where c_{src} was often misclassified as c_{tgt} and vice versa
Table 3.1 shows the setup of the attacks, whose evaluations can be seen in 3.2. The concrete implementation used by [53] can be seen in Algorithm 1. $m\_cnt_{src}^{target}$ indicates the amount of misclassifications in a non-adversarial setup. For information about the labels of the classes, see Section 2.4.

Table 3.1: Attack Configuration and baseline given in [53]

	CIFAR-10		Fashion-MNIST	
setting	$c_{source} \rightarrow c_{target}$	$m\_cnt_{src}^{target}$	$c_{source} \rightarrow c_{target}$	$m\_cnt_{src}^{target}$
I	$5 \rightarrow 3$	15.60%	$6 \rightarrow 0$	23.35%
II	$0 \rightarrow 2$	8.55%	$1 \rightarrow 3$	11.93%
III	$1 \rightarrow 9$	12.10%	$4 \rightarrow 6$	38.10%

Table 3.2: Evaluation Results given in [53]

$c_{source} \rightarrow c_{target}$	$m\_cnt_{src}^{target}$	Malicious Participants						
		2%	4%	10%	20%	30%	40%	50%
CIFAR-10								
$0 \rightarrow 2$	16	1.42%	2.93%	10.2%	14.1%	48.3%	73%	70.5%
$1 \rightarrow 9$	56	0.69%	3.75%	6.04%	15%	36.3%	49.2%	54.7%
$5 \rightarrow 3$	200	0%	3.21%	7.92%	25.4%	49.5%	69.2%	69.2%
Fashion-MNIST								
$1 \rightarrow 3$	18	0.12%	0.42%	2.27%	2.41%	40.3%	45.4%	42%
$4 \rightarrow 6$	51	0.61%	7.16%	16%	29.2%	28.7%	37.1%	58.9%
$6 \rightarrow 0$	118	-1%	2.19%	7.34%	9.81%	19.9%	39%	43.4%

The given percentages refer to the loss in source class recall.

As seen in Table 3.2, both CIFAR-10 and Fashion-MNIST are vulnerable to label flipping attacks, although their levels of vulnerability are not the same. CIFAR-10 is more prone to such attacks - looking at Table 3.2, we can see that the source class recall losses are generally higher on CIFAR-10. The correlation between the $m\_cnt_{src}^{target}$ in a non-adversarial setting versus when under attack is also unclear. As the model for CIFAR-10 already struggles with distinguishing 5 from 3, it would be intuitive to expect the source class recall to drop more than on combinations where the model has higher recalls. This does not seem to be the case - in contrary, the source class recall of setting $0 \rightarrow 2$ (having

the lowest $m\_cnt_{src}^{target}$) dropped the most in four out of six scenarios. In Fashion-MNIST, a similar trend was observed: the $4 \rightarrow 6$ setting experienced larger drops in source class recall than the $6 \rightarrow 0$ setting, despite the latter one having twice as many baseline misclassifications. This indicates that deciding which $c_{source} \rightarrow c_{target}$ combination is the most vulnerable to attack is non-trivial for the attacker, and it doesn't appear that there is a correlation between $m\_cnt_{src}^{target}$ and the effectiveness of the attack [53]. The drop in accuracy of the entire model in general was found to be an order of magnitude smaller than on c_{source} - the attacks appear targeted. This is a desired effect, which means that it is easier for the attack to remain undetected, especially on datasets featuring lots of classes [53].

Algorithm 1 Data Poisoning / Manipulation as seen in [53]

D_{benign}: Benign Dataset

```
function PoisonedDataset(D_benign, c_src, c_tgt)
    D_poisoned ← []
    for sample in D_benign do
        if sample_c = c_src then
            sample_c ← c_target
        D_poisoned.insert(sample)
    return D_poisoned
```

3.1.2 Model Poisoning / Update Manipulation

LIE

The attack proposed in [54] , differs from previous techniques in that it doesn't try to inject updates with maximal disturbance but apply small changes to each parameter. For more details on the calculation of those changes see Algorithm 2 and Section 4.3.2. This makes the attack less effective against averaging aggregation rules, but allows it to stay undetected while targeting TrimmedMean. They also show the ability of this attack to avoid detection by more sophisticated AGR's such as Krum and Bulyan. This attack does not tailor itself to the target AGR [54]. This attack is often used as a benchmark for comparison and is referred to as "LIE Attack" [51].

Fang

Fang et al. proposed multiple attacks for different setups. Those setups differ in the amount of knowledge the malicious clients possess (full knowledge - malicious client knows all local models, partial knowledge - malicious clients only knows his own model). Also, the attack is tailored to the AGR that the server is using. Without this knowledge, the attack has no noticeable impact. The general idea of the proposed attack is to attack the global model by changing the direction of the update gradient towards the inverse direction of the

Algorithm 2 Update Manipulation as seen in [54]

U_{benign}: Benign Update
n: total nodes m: smallest m fulfilling $m + f < n$ (amount of byzantine nodes "missing" to control median)
f: total byzantine nodes

function Z(n, f)
    $m \leftarrow \lfloor \frac{n}{2} + 1 \rfloor - f$
    $z \leftarrow \max_x(\phi(x) > \frac{n-m}{n})$
    return z
function PoisonedUpdate(U_{benign}, z)
    for dim **in** U_{benign} **do**
        $\mu_{dim} \leftarrow \text{mean}(dim)$
        $\sigma_{dim} \leftarrow \text{std}(dim)$
        $P_{poisoned} \leftarrow \mu_{dim} + \sigma_{dim} \cdot z$
    return $D_{poisoned}$
▷ *All malicious nodes now send the same $D_{poisoned}$ for aggregation.* ◁

benign update. Given c malicious worker devices with local model updates $\mathbf{w}'_1, \mathbf{w}'_2, ..., \mathbf{w}'_c$, the compromised workers need to solve the following optimization problem:

$$\max_{\mathbf{w}'_1,...,\mathbf{w}'_c} \mathbf{s}^T(\mathbf{w} - \mathbf{w}'), \tag{3.1}$$

where $\mathbf{s}$ is a column vector of the changing directions of all global model parameters (i.e $s_j = 1$ indicates that the jth global model parameter increases and $s_j = -1$ that it decreases), $\mathbf{w}$ are the weights of the global model before the attack and $\mathbf{w}'$ are the weights of the global model after the attack. [52] The Fang attacks are efficient against synthetic, non-iid datasets. For iid and highly imbalanced non-iid datasets, the performance of the proposed attacks is weak [55].

Table 3.3: Literature Overview (Attacks)

	year	federation	dataset	attack	AGR	malicious
[54]	2019	CFL	MNIST CIFAR-10	Update Manipulation (untargeted)	Krum TrimmedMean Bulyan	5-24%
[52]	2019	CFL	MNIST Fashion-MNIST CH-MNIST	Data Manipulation (untargeted)	Krum TrimmedMean Median	20%
[55]	2021	CFL	CIFAR-10 MNIST FEMNIST	Update Manipulation (untargeted)	Krum TrimmedMean Median Bulyan Multi-Krum	20%
[56]	2018	CFL	MNIST CIFAR-10	Update Manipulation (untargeted)	FedAvg Krum Multi-Krum	30%
[57]	2018	CFL	Fashion-MNIST UCI Adult Census	Data Manipulation (targeted)	Krum	10%
[58]	2020	CFL	CIFAR-10 Fashion-MNIST	Data Manipulation (untargeted)	FedAvg Custom	2-50%

3.2 Defenses against Poisoning Attacks

The following section will give an overview about the different categories of Attacks. Table 3.3 gives an overview about literature that introduces different attacks that belong to the categories listed below.

3.2.1 Aggregation Rules

Due to the byzantian nature of many real-life FL applications, some devices may be compromised and send falsified data. As such, some algorithm or ”rule” is required during the aggregation of model updates to select updates that appear truthful and reject updates that are of malicious nature. In this chapter, a few Aggregation Rules are listed [59].

FederatedAveraging

FederatedAveraging (FedAvg) was introduced by Google in 2016 [1]. The new global model is defined as the average of the model updates of a random subset of all clients. FedAvg performs very bad under attack as even extreme outliers are aggregated as well. FedAvg is effective in non-adversial settings. It is a very efficient aggregation method, as the process of calculating the average requires only little computational power. This efficiency makes it the only AGR viable for large-scale CFL deployments in practice [51].

Algorithm 3 FederatedAveraging (FedAvg)

n: received update vectors
m: size of aggregation set (randomly sampled)

```
1: function FedAvg(n, m)
2:     S_m ← randomly sample m vectors from n
3:     M_aggregated ← empty model, initialized with zeroes
4:     for i in S_m do                                  ▷ i: update
5:         for p, p_index in i do                       ▷ p: parameter
6:             M_aggregated[p_index] ← M_aggregated[p_index] + p
7:     for p in M_aggregated do
8:         p ← p / m
9:     return p
```

Algorithm 3 shows the original implementation as proposed by Google in [1]. The implementation of some papers [60] skip the first step (random sampling of an aggregation set) and average the parameters from all nodes. FedStellar implements the latter version. This is only important in scenarios with large n, where diminishing returns set in and aggregating all nodes provides no noticeable improve in model performance anymore, but leads to higher strain on the network [21].

Krum

Krum[1], introduced in [61] in 2017, tries to identify the most reliable / trusted update. The ranking is decided by a score calculated from the distance to each of the $n - f - 2$ closest vectors to it (n referring to the total amount of updates, f to the amount of potentially byzantine). The distance is calculated as the euclidean distance. This score is calculated for every update received. The vector with the lowest total distance / score is chosen as the new global update. The exclusions of vectors very far by only considerung the $n - f - 2$-closest workers prevents malicious participants from working together by one worker proposing extremely deviated vectors to force the barycenter in the direction of another update proposed by a second malicious participant. Just like Trimmed Mean, Krum will converge if the number of malicious clients is smaller than n. The calculation of Krum is computationally rather expensive, with a runtime of $O(n^2 \cdot d)$ with d being the dimension of the parameter vector, and n being the total amount of nodes [61]. However, the size of the parameter vector influences the runtime only linearly, which is important as in modern ML, d can reach values in the hundreds of billions [62]. A more detailled explanation of Krum can be found in Section 4.3.2.

Multi-Krum

Multi-Krum is a variation of the Krum Aggregation Rule and was introduced in [61]. In Krum, only one proposed update is chosen as the new global model. Multi-Krum

[1]Krum was an emperor of the Bulgarian empire in AD 803-814 who took on many (successful) attacks on the Byzantine empire.

Algorithm 4 Krum

n: received update vectors
f: number of byzantine workers

function KRUM(n)
  for i **in** n **do** ▷ i, j: *update vector*
    $l_{scores}[i] \leftarrow$ SCORE(i, n)
  return MIN(l_{scores})

function SCORE(i, n) ▷ i: *update vector*
  for j **in** n **do**
    $i_{distances}[j] \leftarrow \sum_{m=0}^{len(n)} \|i - n[m]\|^2$
  $i_{distances} \leftarrow sort(i_{distances})$
  $score_i \leftarrow \sum_{m=0}^{n-f-2} i_{distances}[m]$
  return $score_i$

calculates a score for each proposed update using the Krum function. Then, the m best performing updates (the ones with the highest scores) are averaged. If $m = 1$, Multi-Krum behaves exactly like Krum. If $m = n$ with n being the total amount of proposed updates, Multi-Krum just averages all the updates (without applying the Krum function) [63].

Algorithm 5 Multi-Krum

SCORE(i, n): Scoring Function of Krum, as shown in Algorithm 4
FEDAVG(S_m): Federated Averaging function as shown in Algorithm 3
m: amount of updates to select for Krum
n: all received update vectors

function MULTI-KRUM(n, m)
  for i **in** n **do** ▷ i: *update vector*
    $l_{scores}[i] \leftarrow S(i)$
  $l_{selection} \leftarrow sort(l_{scores})[0 : m]$ ▷ *select the m update vectors with the smallest score*
  return $FedAvg(l_{selection})$

Median

The Median AGR was introduced in [64]. Each parameter of the new aggregated model is defined as the median of all submitted weights. If the amount of submitted weights is an even number, the average of the two middle weights is used [64].

Trimmed Mean

Trimmed Mean was introduced in [64]. This rule does not attempt to choose an honest update (like Krum), instead it calculates a new model. First, a list of all weights for

Algorithm 6 Median

n: received update vectors

```
1: function MEDIAN(n)
2:     M_aggregated ← empty model, initialized with zeroes
3:     for i in n do                                   ▷ i: update
4:         for p, p_index in i do                      ▷ p: parameter
5:             M_aggregated[p_index] ← M_aggregated[p_index] + P
6:     for p in M_aggregated do
7:         p ← MED(p)
8:     return p
```

each parameter of the model is composed from all received updates. The β smallest and β largest weights are removed. The weights for each parameter of the new aggregated model is defined by the mean of the remaining $m-2\beta$ weights. β is given as a parameter to the aggregation function [64]. Trimmed Mean is effective against attacks, where malicious vectors will be far away from honest vectors (for example simple data manipulation, such as label flipping). If the number of malicious clients is smaller than β, the global model will converge when confronted with such attacks [54].

Algorithm 7 Trimmed Mean

β: trimming factor

n: received update vectors

```
1: function TRIMMEDMEAN(β, n)
2:     list_params ← []
3:     for i in n do                                   ▷ i: update
4:         for p, p_index in i do                      ▷ p: parameter
5:             list_params[p_index] ← list_params[p_index] + p
6:     for p in list_params do
7:         p ← p[β : −β]          ▷ remove β elements from the start and end of the list
8:         p ← MEAN(p)
9:     return p
```

Bulyan

Bulyan[2] was introduced in [65]. The concept of the Bulyan AGR is a combination of a byzantine–resilient aggregation rule (BAR) and (a variation of) TrimmedMean. It first uses the BAR (the authors run their experiments using Krum, but others such as Brute or Median can be used too) to generate a subset of clients which are (probably) benign. This subset is then aggregated using TrimmedMean, which detects attacks that only try to manipulate a single parameter of the update (for example Backdoor Attacks). Krum alone wouldn't be able to stop such an attack, as it doesn't evaluate each parameter of the update on its own - just the most trusted update is chosen as a whole. TrimmedMean

[2]Bulyan, better known as Count Julian, was a Byzantine general, who betrayed the Byzantine empire by secretly associating with the Muslims. In other words, he was bzyantine to the Byzantines.

however, calculates a new update by aggregating each parameter seperately. For Trimmed Mean to successfully select only honest participants, the subset selected by Krum needs to have a majority of benign clients. Thus, Bulyan is designed to defend against $\frac{m-3}{4}$ (with m being the amount of benign clients) malicious workers [65]. It is also often used as a benchmark for comparing attack performance [55]. However, Bulyan does not scale well - the Krum aggregation is run multiple times in each iteration. Krum itself computes pairwise distances (n^2 operations) between local model updates [52].

Algorithm 8 Bulyan

n: received update vectors
f: amount of malicious clients
A: any (α, f)-Byzantine–resilient aggregation rule, e.g. Krum

function BULYAN(β, n, f)
  $S_n \leftarrow []$
  while LENGTH(S_n) $< (n - 2f)$ **do**
    $n_{rest} \leftarrow n \setminus S_n$
    $S_n \leftarrow S_n + \mathrm{A}(n_{rest})$
  return TRIMMEDMEAN(β, S_n)

Table 3.4: Overview (Aggregation Rules)

	year	AGR	convergence condition	runtime
[21]	2016	FederatedAveraging	-	$\mathcal{O}(n^2 \cdot d)$
[64]	2019	Median	$n > 2f$	$\mathcal{O}(n \cdot d \cdot log^3 \frac{1}{\epsilon})$
[64]	2018	TrimmedMean	$n > 2f$	$\mathcal{O}(n \cdot logn \cdot d)$
[61]	2017	Krum	$n > 2f + 2$	$\mathcal{O}(n^2 \cdot d)$
[63]	2017	Multi-Krum	$n > 2f + 2$	$\mathcal{O}(n^2 \cdot d)$
[65]	2018	Bulyan	$n \geq 4f + 3$	$\mathcal{O}(n^2 \cdot d)$

n refers to the total amount of nodes, f to the amount of byzantine nodes

3.3 Discussion

In 2021, [51] presented an analysis of poisoning attacks in setups which they consider to be more realistic than the ones chosen by other papers. They claim that the attack/defense setups used in other popular papers do not hold in realistic scenarios. For example [54] gives data depicting the loss of accuracy of a model attacked by 5-24% corrupt workers. [52] also test their attacks using 20% compromised devices. [51] claim that in those papers, the number of malicious clients is unrealistically high and the research therefore would not extend to reality. They suggest that 0.1% or 0.01% may be a more realistic assumption. To set these numbers in perspective, they explain that an attacker would have to control 250 million Android devices to control 25% of the nodes using Gboard (a keyboard application from Google using FL). Therefore, they claim Federated Learning is highly robust in practice even when only simple defense methods are used - assuming that the application

is widespread enough to have millions of benign clients [51]. However, in the same year the same author published a paper showcasing a few newly developed attacks (agr-tailored, agr-agnostic, ...). In the evaluations, all adversarial setups were evaluated using 20% malicious clients [55]. The concept of most byzantine-robust Aggregation Rules is based on comparing updates and then trying to remove outliers from the aggregation set. However, in most DFL topologies, the nodes have only 1 or 2 (e.g. Ring) neighbors that they can aggregate their model with. The aggregating node finds itself in a position where the attacker controls half or a third of all updates that are available to him. As can be seen in Table 3.4, most aggregation rules are not able to converge anymore.

Chapter 4

Design & Implementation

In this chapter, the FedStellar code structure is discussed in more detail. After an overview of the FedStellar codebase, key implementation details done during the work of this thesis are discussed. Due to the number of small changes that were done to improve on the functionality / reliability of the framework, only the implementation of larger elements such as Aggregation Rules and Attacks are discussed in this chapter. Code snippets are also included where relevant.

4.1 Design

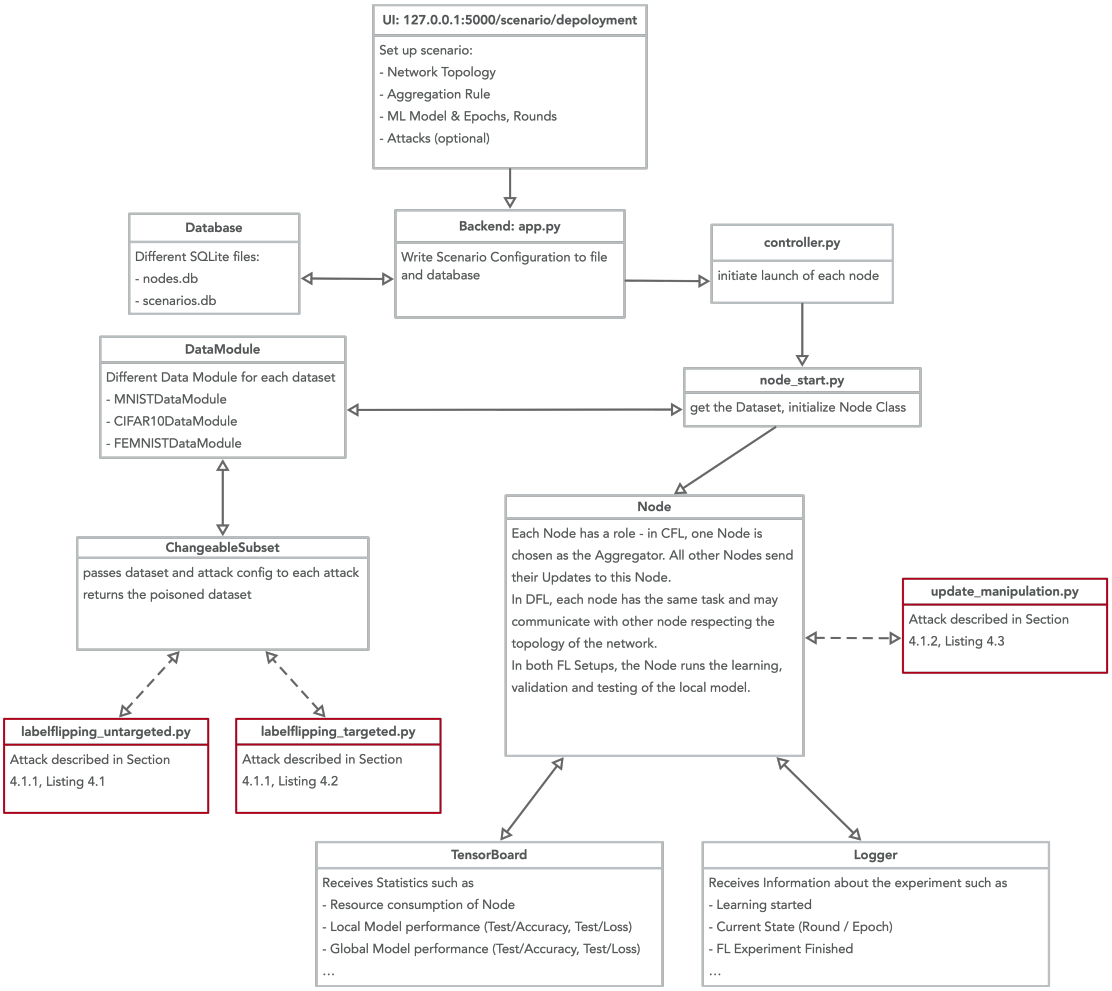


Figure 4.1: FedStellar Architecture

Figure 4.1 shows the general architecture of FedStellar, as well as the execution of the different attacks. Note that for this thesis, the nodes were not deployed, but all ran on the same machine (”Simulation”). After deploying a scenario from the webinterface, the webserver will store the created scenario in the database, and launch the controller. The controller is used to collect all necessary configuration parameters and launch the nodes. For each node, ”node_start.py” is executed in a seperate process detached from the Backend. Each node is assigned a port, on which the node communicates. After launching, the node first calls the DataModule that he was initialized with - each dataset has its own DataModule (e.g. MNISTDataModule). The DataModule doesn’t load the entire dataset for each node, only a part depending on the configuration. Also, the Data Manipulation attacks are executed here, denoted by the two red boxes on the left of Figure 4.1. The Data Manipulation attacks change the dataset according to the parameters (principle given in Algorithm 1, implementation in Listing 4.5, 4.4, 4.6). The attacking node does not behave any different from a benign node, it just gets a manipulated dataset to train on. The Update Manipulation attacks (principle given in Algorithm 2, implementation in

Listing 4.3.2) run on the node directly, denoted by the red box on the right of Figure 4.1. Finally, each node maintains a connection to TensorBoard [1]. All kind of statistical data, be it Resource consumption, Model metrics such as Test/Accuracy or Test/Loss and the training progress is aggregated and displayed in FedStellars interface.

4.2 FedStellar Interface

Configuring a FL scenario using FedStellar is very simple. In the default "User mode" (shown in 4.2), a basic non-adversarial setup using sensible default values can be run.

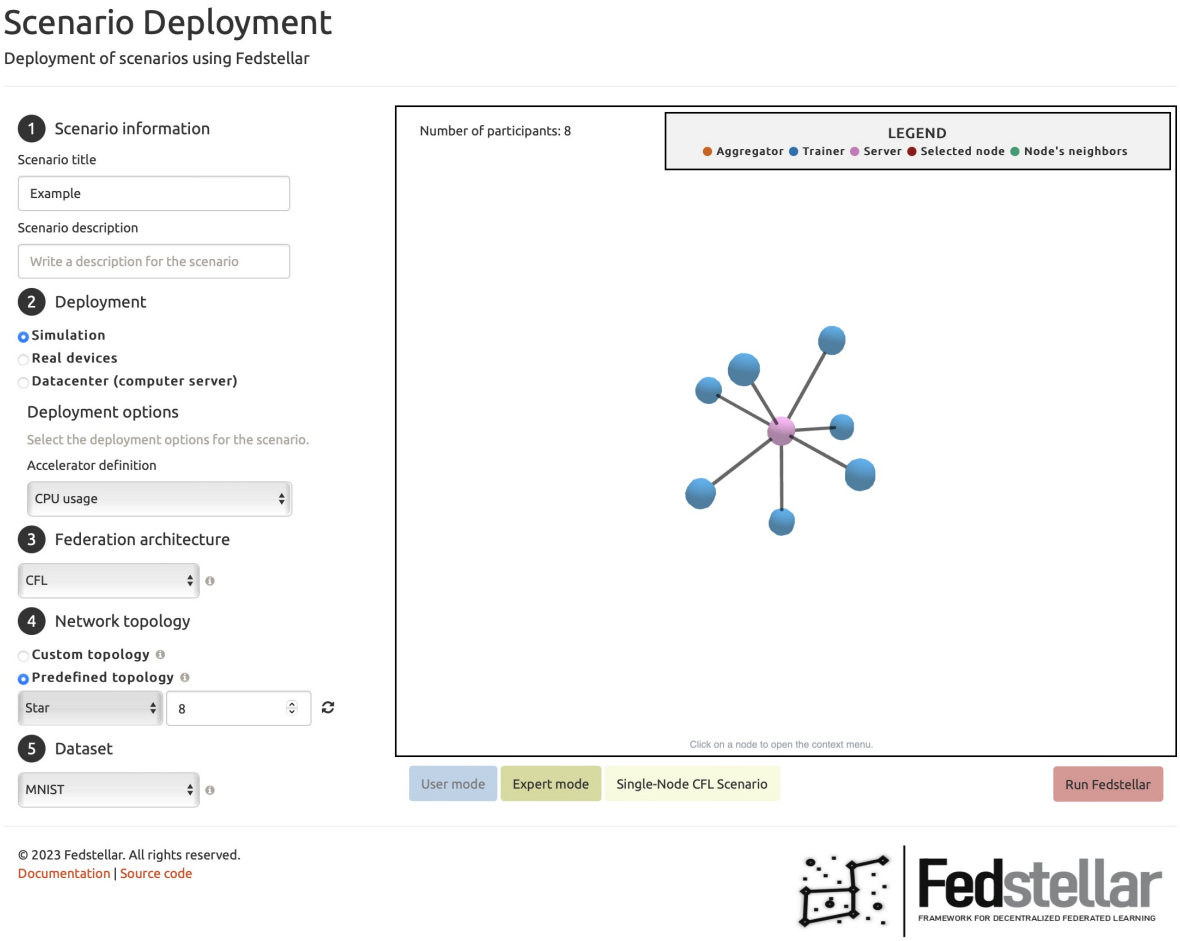


Figure 4.2: FedStellar Scenario Setup

For further control of the learning process (epochs, rounds) as well as deploying adversarial setups, a second set of controls is available ("Expert mode", shown in in 4.3). Note that in the FedStellar version of this thesis, this mode is enabled by default.

[1] TensorBoard is the vizualization toolkit of Tensorflow: a suite of web applications for inspecting, evaluating and exporting statistical data from model training processes [66].

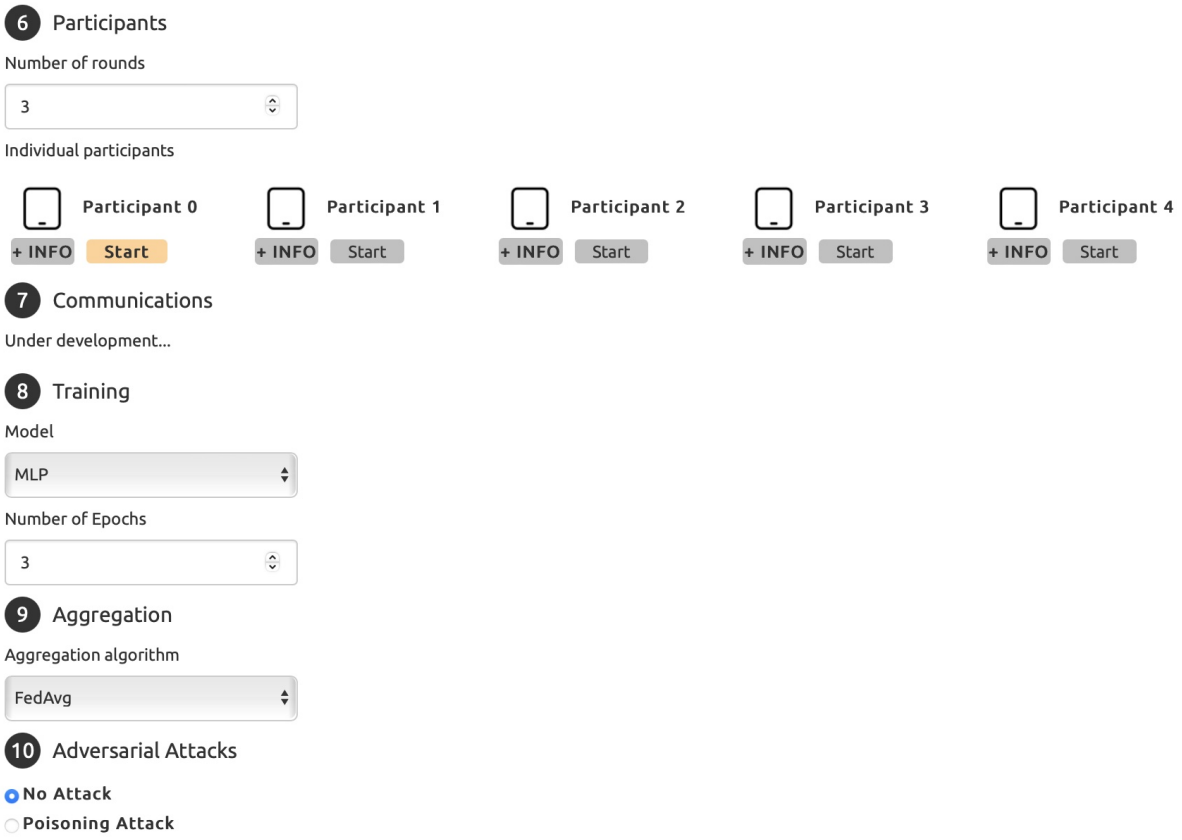


Figure 4.3: FedStellar Scenario Setup: Additional Configuration

If a user selects a poisoning attack, he can set up all the required attack parameters in the interface as well. Note that the targeted data manipulation attacks support the selection of multiple source classes. The original implementation of those attacks do not provide this option - thus the given implementation in python does not correspond 1-to-1 to the equivalent Pseudocode (see Algorithm 1).

Figure 4.4: Configuration: Data Manipulation - targeted, unspecific

Figure 4.5: Configuration: Data Manipulation - targeted, specific

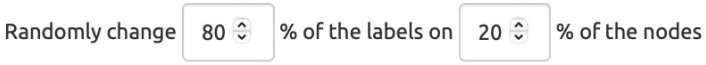


Figure 4.6: Configuration: Data Manipulation - unspecific

For setting up the Update Manipulation (LIE) Attack, the user needs to find the optimal z. Clicking the Button seen in 4.7 will calculate z from the scenario configuration as described in [54]. Not calculating z on the attacking node itself is an intentional decision, the user setting up the scenario may decide not to use the optimal z (using a smaller z than calculated can be used to account for inaccuracies in the calculation of μ and σ (see Algorithm 2) [54]).

Figure 4.7: Configuration: Update Manipulation - LIE

4.3 Implementation: Attacks

4.3.1 Data Manipulation, targeted

The following targeted attacks aim to maximise the accuracy reduction of the global model on one (or more) given labels by manipulating the training data. The application code, model and update processes are not changed by these attacks [67].
Listing 4.1 shows a version that replaces the given label with a new label, that is randomly chosen from all available classes (excluding the label itself).
Listing 4.2 shows a more specific version of the attack. Here, the attacker not only specifies which label(s) are attacked, but also to what they are changed. This can be used to target certain label combinations where the model performs inherently worse, for example 3 and 8 in the MNIST dataset [68].
These attacks are efficient in scenarios, where the aggregation rule does not filter outliers (such as FedAvg). When aggregation methods such as Trimmed Mean are used, such

attacks have little to no effect as the malicious updates are easily detected as statistical outliers. If those Aggregation methods need to be avoided, more sophisticated attacks such as the one shown in Listing 4.3 can be used [54].

```
def labelflipping_targeted_unspecific(dataset, indices, label_og: int):
    """
    This attack changes the label given in label_og to some other
    label (selected randomly)
    :param dataset: Dataset to flip the labels of
    :param indices: Indices of subsets where the attack will be applied
    :param label_og: The label / class ID which will be changed to some
                     other label (selected randomly)
    :return: Manipulated Dataset
    """
    new_dataset = copy.copy(dataset)
    targets = new_dataset.targets.detach().clone()
    class_list = new_dataset.class_to_idx.values()

    for i in indices:
        t = targets[i]
        if t == label_og:
            targets[i] = torch.tensor(
                random.sample(
                    sorted(
                        [x for x in class_list if x != t]
                    ), 1)
            )

    new_dataset.targets = targets
    return new_dataset
```

Listing 4.1: Labelflipping Attack: targeted; unspecific

```
def labelflipping_targeted_specific(dataset, indices, label_og: list,
    label_goal: int):
    """
    This attack changes the label(s) given in label_og to the label given
        in label_goal
    :param dataset: Dataset to flip the labels of
    :param indices: Indices of subsets where the attack will be applied
    :param label_og: The original label(s) / class ID(s) which will be
        changed to label_goal
    :param label_goal: The label / class ID to which label_og will be
        changed
    :return:
    """
    new_dataset = copy.copy(dataset)
    targets = new_dataset.targets.detach().clone()
    for i in indices:
        t = targets[i].numpy()
        if (t in label_og) or (str(t) in label_og):
            targets[i] = label_goal
    new_dataset.targets = targets
    return new_dataset
```

Listing 4.2: Labelflipping Attack: targeted; specific

4.3.2 Update Manipulation

The following attack aims to reduce the global model accuracy without targeting a specific class (untargeted). This attack tries to manipulate the updated parameter as much as possible while avoiding being detected. It does so by applying small changes to all parameters. This attack is tailored to AGRs that try to reject outliers such as Trimmed Mean or Krum. When attacking other AGRs like FedAvg (that do not reject statistical outliers) other attacks are more efficient and the "gentle" approach of the LIE attack is not needed. In order to manipulate the global model aggregated by TrimmedMean, the attacker needs a majority of workers (to be in control of the median). To achieve this, the attacker first calculates the amount of workers missing for a majority (s). Then, the attacker can use the normal distribution ($\phi(z)$) to determine the value of z_{max} at which (s) are likely farther from the mean. For calculation of the new parameters to form an update, the attacker needs to calculate the mean (μ) and standard deviation (σ) of the distribution. Then, the range of values can be calculated: $(\mu - z\sigma, \mu + z\sigma)$

By assigning the malicious nodes values in this range, the attacker ensures with high probability that those values set the median. Also, due to all malicious nodes reporting those values, the malicious values will survive the averaging of the remaining nodes (after the cutoff of the TrimmedMean AGR).

For example, in a scenario with 100 workers and 47 malicious clients, the attacker would need $(\frac{100}{2} + 1) - 47 = 4$ additional clients to control the median. According to the z-table, the maximal z (z_{max}) for which $\phi z < \frac{100-4}{100} = 0.96$ is 1.75. For each parameter x of the benign model, the attacker will now calculate a new one (x'):

$$x' = \mu[x] + \sigma[x] \cdot z_{max}$$

Listing 4.4 showcases the implementation of the calculation of z_{max} in Python. Please note that this function is not executed by the attacking node itself, but is a function of the backend exposed as an API (app.py, see Figure 4.1). It is called from the frontend when the user selects the attack parameters (such as the % of malicious nodes). This allows the change of the parameter before running the experiment. In practice, this calculation could run on the node itself. In the context of the FedStellar Framework, each node is aware of the total amount of nodes as well as the amount of malicious nodes. [54] reduced the z_{max} value by a bit before running the experiment (to 1.5 from 1.75) in order to account for inaccuracies in the estimation of μ and σ [54].

```
1  def update_manipulation_LIE(parameters, z):
2      """
3      :param parameters: Honest parameters (calculated by client)
4      :param z: by how many standard deviations the parameter gets skewed
5      :return: Malicious parameters
6      """
7      malicious_parameters = {}
8      for key, value in parameters.items():
9          if key.endswith("bias"):
10             malicious_parameters[key] = value
11         else:
12             new_weights_list = []
13             for weights in value:
```

```
                new_weights = []
                avg = torch.mean(weights, dim=0)
                std = torch.std(weights, dim=0)
                for _ in weights:
                    new_weights.append(avg + z \cdot std)
                new_weights_list.append(new_weights)
            malicious_parameters[key] = torch.tensor(new_weights_list)
    return malicious_parameters
```

Listing 4.3: Update Manipulation Attack: LIE

```
def find_zmax(percent_malicious: int, total_nodes: int):
    malicious_nodes = int(math.ceil(
            total_nodes * (percent_malicious / 100)
        )
    )
    # if malicious_nodes > total_nodes, the median is already under control
        of the attacker and convergence of the global model is no longer
       possible -> the attacker is free to choose any z he chooses
    if malicious_nodes > math.ceil(total_nodes/2):
        return 999
    nodes_required_for_majority = math.ceil(
        ((total_nodes / 2) + 1) - malicious_nodes
    )
    # ppf = percent point function (inverse of cdf)
    z_max = scipy.stats.norm().ppf(
        (total_nodes - nodes_required_for_majority) / total_nodes
    )
    z_max_rounded = math.floor(z_max * 100) / 100.0
    return z_max_rounded
```

Listing 4.4: LIE Attack: Calculation of z_{max}

4.4 Implementation: Aggregation Rules

4.4.1 Krum

The Krum AGR is already given by the FedStellar framework. However, the implementation did not correspond to the one defined in [61], where the score is defined as

$$score_i \leftarrow \sum_{m=0}^{n-f-2} i_{distances}[m] \tag{4.1}$$

In other words, the sum of the distances of the $n - f - 2$-nearest nodes to the score. In FedStellar the score is defined as

$$score_i \leftarrow \sum_{m=0}^{n} i_{distances}[m] \tag{4.2}$$

This approach however does not tolerate more than one byzantine worker, as one attacker could propose very large vectors pulling the barycenter towards a vector that is proposed

by the second attacker, which would then be chosen as the global model. Thus, the give implementation was modified to correspond to the original algorithm proposed by [61]. The corrected implementation is shown in Listing 4.5.

```
    def aggregate(self, models):
        """
        Args:
            models: Dictionary with the models (node: model,num_samples).
        """
        # Check if there are models to aggregate
        if len(models) == 0:
            logging.error("[Krum] No models available for aggregation")
            return None

        models = list(models.values())

        # Create a Zero Model
        accum = (models[-1][0]).copy()
        for layer in accum:
            accum[layer] = torch.zeros_like(accum[layer])

        logging.info(f"[Krum] Aggregating models: num={len(models)}")

        # Create model distance list
        total_models = len(models)
        distance_list = {k: [] for k in range(0, total_models)}

        # Calculate the L2 Norm between all models
        for i in range(0, total_models):
            m1, _ = models[i]
            for j in range(0, total_models):
                m2, _ = models[j]
                if i != j:
                    for layer in m1:
                        l1 = m1[layer]
                        l2 = m2[layer]
                        distance_list[i].append(numpy.linalg.norm(l1 - l2))

        n = int(self.config.participant["scenario_args"]["n_nodes"])
        node_percent = self.config.participant["adversarial_args"]["
            node_percent"]
        f = int(math.ceil(node_percent * n))

        # score = sum of distance to n-f-2 nearest nodes
        score_list = {}
        for i, distances in distance_list.items():
            sorted_distances = sorted(distances)  # ascending
            n_f_2_smallest = sorted_distances[: n - f - 2]
            score_list[i] = sum(n_f_2_smallest)

        # Assign the model with min score as the chosen model
        chosen_model, _ = models[min(score_list, key=score_list.get)]
        print("[Krum] Selected Model: ")
        print(min(score_list, key=score_list.get))
        for layer in chosen_model:
            accum[layer] = accum[layer] + chosen_model[layer]

```

```
53          return accum
```

Listing 4.5: Krum AGR

4.4.2 Bulyan

The Bulyan AGR is, as seen in Algorithm 8, a combination of TrimmedMean and another Byzantine-robust AGR ([65] propose Krum). Krum is used to generate a set of probably benign updates, which are then aggregated coordinate-wise by TrimmedMean. For more details, see Algorithm 8. As [65] suggest using Krum, this was chosen for the implementation.

```
1     def aggregate(self, models):
2         if len(models) == 0:
3             logging.error("[Bulyan] Trying to aggregate models when there
                  is no models")
4             return None
5
6         # Krum Step
7
8         models = list(models.values())
9         if len(models) == 1:
10            accum = (models[-1][0]).copy()
11            for layer in accum:
12                accum[layer] = torch.zeros_like(accum[layer])
13            return accum
14
15        # initialize zeroed model
16        accum = (models[-1][0]).copy()
17        for layer in accum:
18            accum[layer] = torch.zeros_like(accum[layer])
19
20        logging.info("[Krum.aggregate] Aggregating models: num={}".format(
              len(models)))
21
22        # Create model distance list
23        total_models = len(models)
24        distance_list = {k: [] for k in range(0, total_models)}
25
26        # Calculate the L2 Norm between all models
27        for i in range(0, total_models):
28            m1, _ = models[i]
29            for j in range(0, total_models):
30                m2, _ = models[j]
31                if i != j:
32                    for layer in m1:
33                        l1 = m1[layer]
34                        l2 = m2[layer]
35                        dist = numpy.linalg.norm(l1 - l2)
36                        if dist != 0:
37                            distance_list[i].append(numpy.linalg.norm(l1 -
                                  l2))
38
```

```
39          # calculate n-f-2
40          print("CONFIG")
41          print(self.config.participant)
42          n = int(self.config.participant["scenario_args"]["n_nodes"])
43          node_percent = self.config.participant["adversarial_args"]["
                node_percent"]
44          f = int(math.ceil(node_percent * n))
45
46          # calculate the score as the sum of distance to the n-f-2 nearest
                nodes
47          print(distance_list)
48          score_list = {}
49          for i, distances in distance_list.items():
50              sorted_distances = sorted(distances)  # Sort the list in
                    ascending order
51              n_f_2_smallest = sorted_distances[: n - f - 2]
52              score_list[i] = sum(n_f_2_smallest)
53
54          # Order the score list by score
55          score_list_ordered = dict(sorted(score_list.items(), key=lambda
                item: item[1]))
56
57          # remove the potentially malicious models
58          if len(score_list_ordered) <= self.KRUM_SELECTION_SET_LEN:
59              logging.error(
60                  "[Bulyan(TRMstep)] Trying to aggregate models when there
                        are less or equal models than the set length of the krum
                        selection set ..."
61              )
62              return None
63          else:
64              probably_benign_models = score_list_ordered[: self.
                    KRUM_SELECTION_SET_LEN]
65
66          # TrimmedMean Step
67
68          models = [x[1] for x in probably_benign_models]
69          TRM = TrimmedMean(beta=self.TRM_BETA)
70          return TRM.aggregate(models)
```

Listing 4.6: Bulyan AGR

Chapter 5

Evaluation

This chapter is structured in two sections. In the first section, the behaviour and limits of the attacks in CFL scenarios is evaluated. The second section focuses on DFL, especially different network topologies.

The summaries of the models (provided by FedStellar) used in this evaluation are available in Appendix C. If not stated otherwise, n refers to the total amount of nodes and f to the amount of byzantine nodes. In CFL scenarios, the amount of trainers is equal to $n - 1$.

5.1 Attack Behaviour (CFL)

Scenarios in this section use, if not stated otherwise, the MNIST dataset with the MLP model (summary in Table C.2).

Targeted Data Manipulation

In this subsection, the targeted Attack in CFL is investigated. [53] presented Data Poisoning attacks on Fashion-MNIST. When working with 20% malicious nodes, they saw a 30-40% reduction in source recall on the targeted class, depending on the dataset and label combination used. Results are given for Fashion-MNIST as well as CIFAR-10. The reductions in source recall depend on the dataset, CIFAR-10 saw higher declines in general due to the inherent complexity of the classifying task [53].

Using the settings of the experiment shown in [53], a FedStellar scenario was created to compare the results shown in [53] to the MNIST dataset. The scenario uses CFL, $n = 5$, $f = 1$, $c_{src} = 1$, $c_{target} = 7$. Figure 5.1 shows the confusion matrix of the global model after aggregation. While the source class recall drops by ≈30%, the global model accuracy only dropped by ≈3%. The observed results are very similar to the results given in [53].

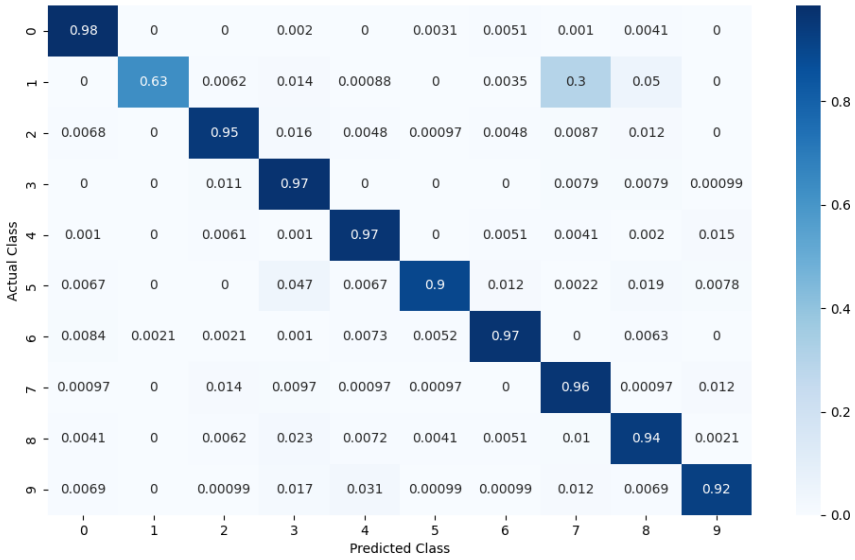
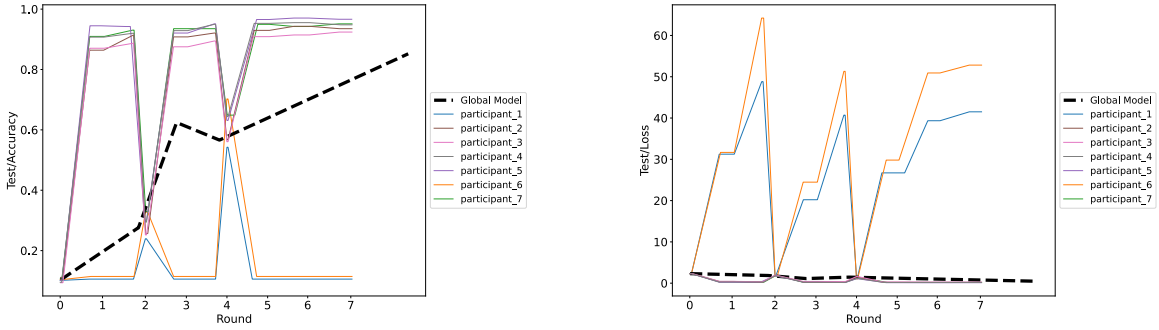


Figure 5.1: Targeted Data Manipulation: Global Model Confusion Matrix

Limits of Data Manipulation and FedAvg

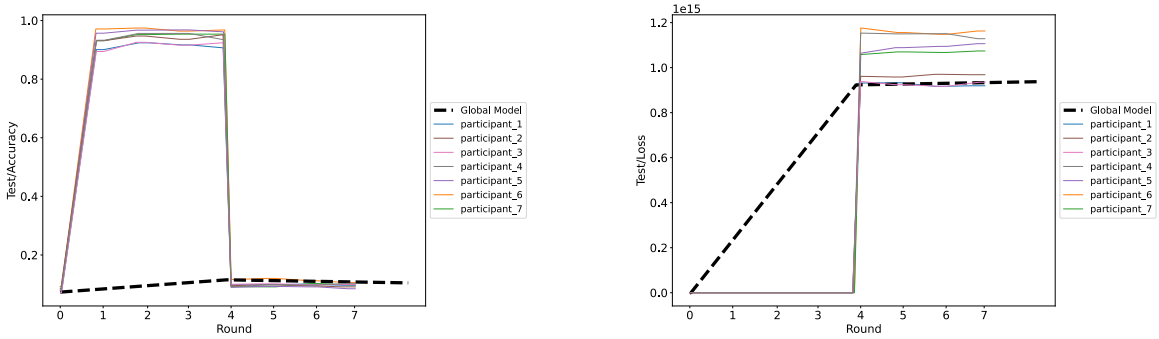
Figure 5.2 shows a scenario, in which 2 of the 7 training nodes are controlled by the attacker (28.5% malicious nodes). The attacking nodes replace all labels with '0', achieving a local model accuracy of ≈10%. Despite that, the global model still converges after just a few training rounds, the final global model accuracy after four aggregations is 85.2%.
If a attacker is aware that the server is using FedAvg, he can use a tailored AGR Attack. Figure 5.3 shows the same setup, but to further illustrate the problem with FedAvg just one of 7 training nodes is controlled by the attacker. The attack used in this setup is identical to the LIE attack in 4.3.2, but line 18 was replaced by `new_weights.append(9999)` to exploit the fact that FedAvg does not remove statistical outliers before aggregation. If FedAvg is attacked like this, the global model will not converge, no matter how few nodes are malicious. This tailored attack can be easily stopped by applying a AGR, that removes outliers before aggregation. In this context, it doesn't matter whether the checking for outliers happens on basis of the entire update vector (e.g. Krum) or separately for each parameter (e.g. TrimmedMean). Figure 5.4 shows the same attack shown in Figure 5.3 but combined with TrimmedMean. Now, the attack has no effect on the global model anymore as his updates are no longer taken into account when aggregating. In theory, TrimmedMean can be circumvented by applying update manipulation attacks as described in [54] (see Algorithm 2).
However, efforts to reproducing the results given in [54] were unsuccessful. Upon further investigation, some potential reasons have been discovered. In general the Attack of [54] is based on the concept of estimating how large a malicious update can deviate from the benign ones, but staying just small enough to be undetected. For a successful attack, the attacker needs to run the attack for many rounds and control a large amount of nodes. [54] experimented with 24% malicious clients on MNIST. Still it took 25 rounds to reach

a 2% loss in accuracy when targeting TrimmedMean. When evaluating MNIST scenarios in FedStellar with $f = 0$, the test accuracy of the participants deviate by more than 2% already. By checking logs, it is possible to know whether the malicious parameters were excluded from the aggregation set by the AGR, but it would be impossible to tell whether a drop in accuracy is just a usual deviation, or the result of a successful attack. Also, limitations of the hardware available for running the evaluations make it impossible to simulate scenarios to more than 100 rounds, where larger (>10%) accuracy reductions are expected. According to [54], CIFAR-10 would be much simpler to attack - significant (>15%) accuracy losses have been observed after 100 rounds [54]. However, on the hardware used for evaluation running a CIFAR-10 training round with one epoch and 5 nodes takes ≈17 minutes. Therefore, it was not feasible to run any evaluations on CIFAR-10.



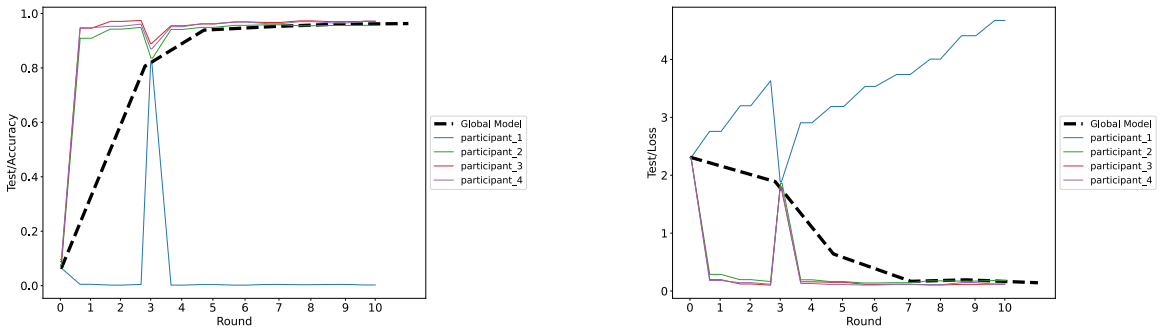
(a) Test/Accuracy (b) Test/Loss

Figure 5.2: Data Manipulation: $n = 8$, $f = 2$, FedAvg



(a) Test/Accuracy (b) Test/Loss

Figure 5.3: Update Manipulation: $n = 8$, $f = 1$, FedAvg



(a) Test/Accuracy (b) Test/Loss

Figure 5.4: Update Manipulation: $n = 5$, $f = 1$, TrimmedMean

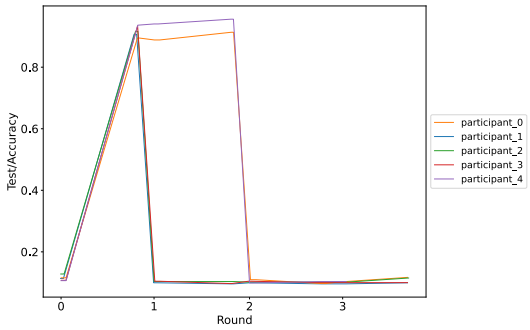
5.2 Attack Behaviour (DFL)

Propagation of an Attack

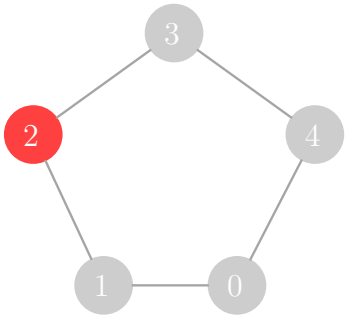
The propagation of the attack through the network depends on the network topology. In a Fully connected network, the honest nodes receive malicious updates in the first aggregation round, and after that they aggregate with new malicious parameters every round again. On the other hand, in a Ring network, the malicious parameters are aggregated (which means that ideally, they get skewed to the honest parameters) and then sent to the next node receives the already once aggregated malicious parameters for the next training round. These propagation processes are illustrated in Figure 5.8. For all scenarios in this chapter, the update manipulation attack tailored to FedAvg was used to make the statistic graphs as clear as possible.

Figure 5.5 shows the FedStellar statistics for the Ring topology, when confronted with a single attacker (participant_2). Note that the first measurement of Test/Accuracy of participant_2 is >90% because the first data point is logged to Tensorboard before the first aggregation. At this point, the parameter have not been manipulated yet (this happens only when they are sent for aggregation). With this in mind, Figure 5.5 shows the process illustrated in Figure 5.8 (c) clearly: In the first aggregation round, participant 1 and 3 are aggregated, and in the second round, participant 4 and 0 follow. Figure 5.7 shows the propagation of the attack as shown in Figure 5.8(a) and (b). Figure 5.7 (a) shows the star topolgy when a leaf (participant 1) is poisoned. The center node (participant 0) has to aggregate twice for the other leaves to become affected. Figure 5.7 (b) shows that if the center node is the adverary, all leaves are affected from the first round on.

As shown in Figure 5.8, in a fully connected Scenario, the attack will be applied to all nodes already in the first aggregation round.

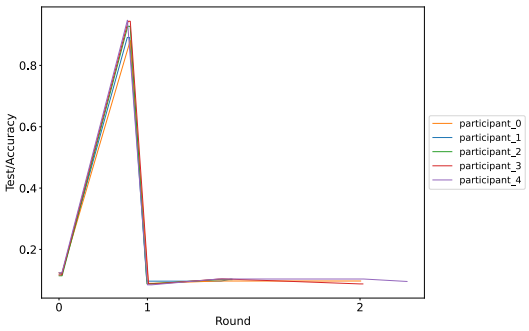


(a) Test/Accuracy



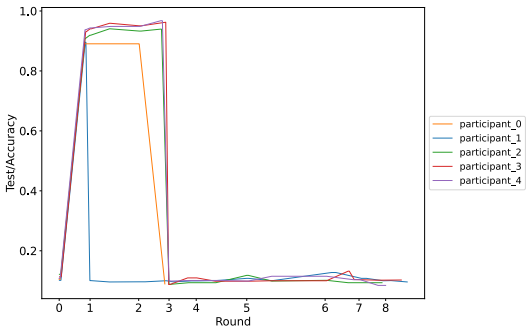
(b) Topology

Figure 5.5: Attack Propagation Ring: $n = 5,\ f = 1$

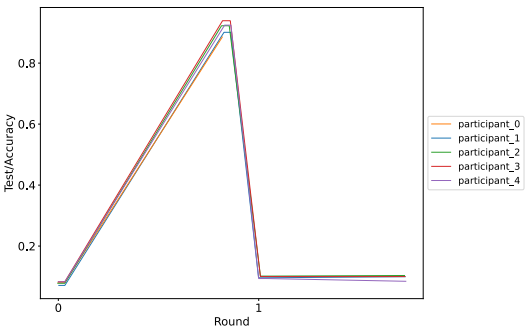


(a) Test/Accuracy

Figure 5.6: Attack Propagation Fully Connected: $n = 5,\ f = 1$



(a) Poisoned Leaf



(b) Poisoned Center Node

Figure 5.7: Attack Propagation Star: $n = 5,\ f = 1$

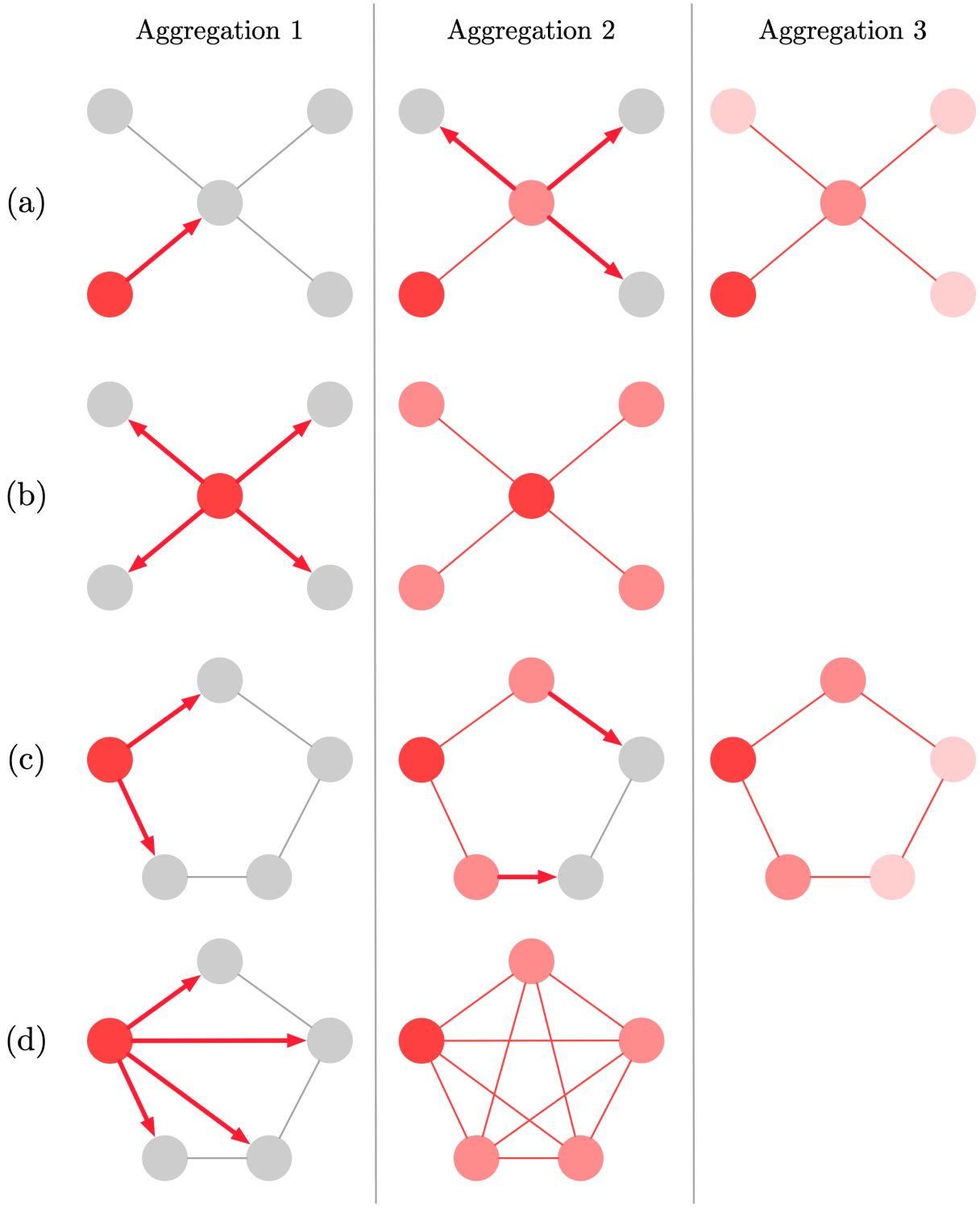
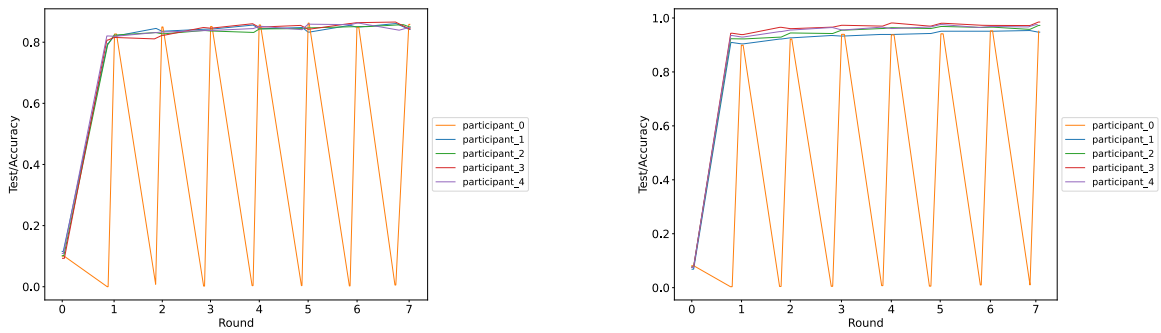


Figure 5.8: Attack Propagation in different Network Topologies

Fully Connected

In this section, all experiments were run using MNIST with MLP model, 7 rounds and 1 epoch each. Figure 5.9 shows two scenarios with identical settings except the network topology. For these evaluations $n = 5$, $f = 1$ and the adversarial node runs an untargeted labelflipping attack that changes 100% of the labels to a randomly selected another one (excluding itself). Looking at Figure 5.8, we can see that in the Fully connected (d) topology, every node is aggregating its node with the malicious node (as every node a direct neighbor). In the Ring (c) topology, the attack has to aggregate with benign nodes twice to reach all nodes. The impact of the attack is being "diluted" on the way, the further away a node is from the malicious node, the better his performance. Looking at the Test/Accuracy in Figure 5.9(b), we can see that node 3 and 2 have the best performance. Figure 5.10 shows the topology of the scenario, where we can see that those nodes are also the ones that are furthest away from participant 0 (the attacking node).

To evaluate the impact of the attack on the entire scenario, the average Test/Accuracy of all benign nodes is calculated. Here, the scenario using the Fully Connected topology (Figure 5.9(a)) achieves 84.61%. The the scenario using the Ring topology (Figure 5.9(b)) Ring and Fully connected, comes out to 97.01%. In a non-adversarial scenario, the model reaches accuracy of >99%. We can see, that the impact of the attack on the Fully Connected Setting is much higher than on the Ring Setting. The reason for this is illustrated in Figure 5.8 (c) and (d): In the Ring topology, the malicious parameter that get aggregated by participant 3 and 2 (see Figure 5.10) been already "diluted" by the FedAvg once, and are now averaged with benign parameters a second time. In the Fully connected setting, the full effect of the attack already sets in after the first aggregation.



(a) Fully Connected

(b) Ring

Figure 5.9: Fully Connected, Ring: $n = 5$, $f = 1$

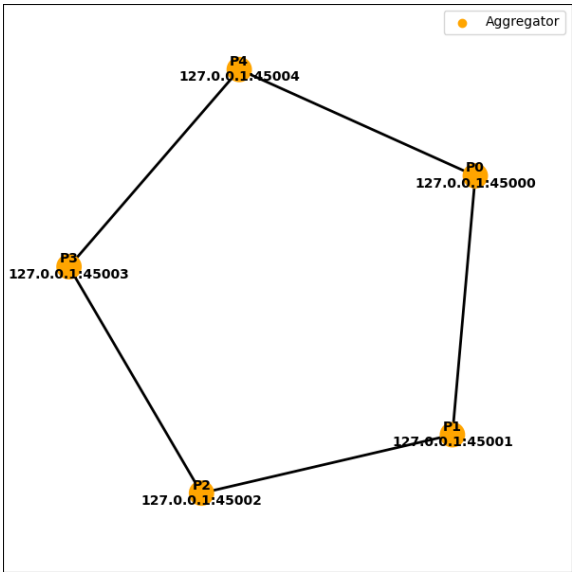


Figure 5.10: Ring Evaluation Network Topology

Network Load

In this section, the consumption of network resources is evaluated. Table 5.1 shows the amount of Bytes and Packets sent and received between 2 training rounds in each scenario. It should be noted that the size of the parameters of the MLP model used in this evaluation account to ≈ 1 MB (see C.2). In the given scenarios, the amount of bytes sent in the Aggregation process in the Ring topology should amount to $size_{parameters} \cdot 2n$, and in the Fully connected topology to $size_{parameters} \cdot (n^2 - n)$. Looking at Table 5.1 it is clearly visible that the Fully connected Topology uses much more network resources, especially when the network grows in size. Adding 4 nodes more than quadrupled the network load (bytes) and the amount of packets transferred increased tenfold.

Note that the measured data is subject to some inaccuracy due to inherent measurement challenges: For the network statistics, FedStellar uses `psutil.net_io_counters()`. This function returns "system-wide network I/O statistics" [69]. As the experiments were run in FedStellar's Simulation mode, every node is running on the host system. Therefore, FedStellar doesn't measure the network load of the scenario, but the consumption of the entire host system. Measuring individual nodes is also not possible for now. Also, the measurements could be influenced by other processes unrelated to the FL scenario. To limit external influence as much as possible, the host system was disconnected from WiFi/LAN. The influence of other local processes on the statistics was measured by using the aforementioned function on the idling, disconnected host system over the observed runtime of the scenarios - these measurements exhibited large deviations (≈ 1.5 MB, 1000 Packets ± ≈75%). Still, the quadratic / linear scaling of the network load is clearly recognizable.

Table 5.1: Network Load Evaluation Scenarios

	Topology	Nodes	Bytes [in MB]		Packets	
			Sent	Received	Sent	Received
120	Fully connected	6	57,85	57,85	14'925	14'916
121	Ring	6	19,34	19,34	5'326	5'344
125	Fully connected	10	279,03	279,01	114'993	115'018
126	Ring	10	34,29	34,30	14'175	14'167

5.3 Discussion

The evaluations shown in Section 5.1 show the influence of a targeted data manipulation attack in a CFL setting. The results given in the confusion matrix (Figure 3.1) align with the findings in [53]. In the next section, the limits of data manipulation and FedAvg were demonstrated. The attacks and aggregation rules performed as expected. The same can be said about the following section analyzing the propagation of the attack in different network topologies. Due to implementation details of the aggregation mechanism, the aggregation of some nodes may not always happen at the same time. If this is kept in mind when analyzing the evaluation results, all performed experiments align with the expectations set by [53], [54] and [2]. The network load statistics also manage to capture the relationship between increasing amount of nodes and network load well, despite the inherent inaccuracies.

Chapter 6

Summary, Conclusions and Future Work

6.1 Summary and Conclusions

In this thesis, the FedStellar framework was expanded with additional functionality to allow simulating various attacks and defenses on both the CFL as well as the DFL setting. This functionality was then used to evaluate DFL settings on their robustness as well as comparing them to CFL. After the necessary background knowledge in the general concepts of Machine Learning and Federated Learning were obtained, a thorough literature review was done on currently used poisoning attacks on FL systems as well as suitable defense mechanisms. Following this, the FedStellar framework was examined in detail and subsequently expanded with additional attacks and defenses. The implemented attacks included data manipulation attacks as well as an update manipulation attack. On the defense side, a new aggregation rule was implemented. Moreover, new datasets were included in the FedStellar framework. The web interface was also expanded to allow users to select and change parameters of the new attack scenarios. Along the way, various issues that plagued the performance or usability of the framework were fixed.

Following the implementation, a number of experiments were conducted, comparing the CFL setting with the DFL setting. For the DFL setting, an additional study was done on the behaviour and robustness of various network topologies. The evaluations yielded the results that were expected, all unexpected results can be explained with the way the learning process is implemented in FedStellar.

6.2 Future Work

As noted earlier, there are several areas where further work is possible. There still exist further aggregation rules and attack techniques (such as backdoor attacks) that are not yet available in FedStellar. Also, the general stability and reproducability of scenarios could be improved. To improve the reproducability of the scenarios, some changes to FedStellar could be made. For example, a globally synchronized aggregation step would

give more consistent results than the way it is implemented now. Also, various avenues for future evaluations and experiments with FedStellar exist. These include (but not limited to) additional experiments further comparing the CFL and DFL architectures under different conditions, the various network topologies and how they impact the speed of model convergence, their susceptibility to malicious clients and impact depending on the position they occupy inside the topology (e.g center node or leaf node in a star topology), etc. Additionally, running the experiments on a more powerful system than the host system (CPU-only) used for the evaluations would enable testing of larger CNN models on more complex datasets such as CIFAR-10 or CIFAR-100, optionally in conjunction with a large number of nodes.

Bibliography

[1] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, "Federated learning of deep networks using model averaging", *CoRR*, vol. abs/1602.05629, 2016. arXiv: 1602.05629. [Online]. Available: http://arxiv.org/abs/1602.05629.

[2] E. T. M. Beltrán, M. Q. Pérez, P. M. S. Sánchez, *et al.*, *Decentralized federated learning: Fundamentals, state-of-the-art, frameworks, trends, and challenges*, 2023. arXiv: 2211.08413 [cs.LG].

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, p. 436, 2015.

[4] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: 1502.03167 [cs.LG].

[5] L. Huang, J. Qin, Y. Zhou, F. Zhu, L. Liu, and L. Shao, "Normalization techniques in training dnns: Methodology, analysis and application", *CoRR*, vol. abs/2009.12836, 2020. arXiv: 2009.12836. [Online]. Available: https://arxiv.org/abs/2009.12836.

[6] P. Cerda, G. Varoquaux, and B. Kégl, "Similarity encoding for learning with dirty categorical variables", *CoRR*, vol. abs/1806.00979, 2018. arXiv: 1806.00979. [Online]. Available: http://arxiv.org/abs/1806.00979.

[7] J. Qi, J. Du, S. M. Siniscalchi, X. Ma, and C.-H. Lee, "On mean absolute error for deep neural network based vector-to-vector regression", *IEEE Signal Processing Letters*, vol. 27, pp. 1485–1489, 2020. DOI: 10.1109/lsp.2020.3016837. [Online]. Available: https://doi.org/10.1109%2Flsp.2020.3016837.

[8] K. Janocha and W. M. Czarnecki, *On loss functions for deep neural networks in classification*, 2017. arXiv: 1702.05659 [cs.LG].

[9] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[10] R. Gu, S. Yang, and F. Wu, "Distributed machine learning on mobile devices: A survey", *CoRR*, vol. abs/1909.08329, 2019. arXiv: 1909.08329. [Online]. Available: http://arxiv.org/abs/1909.08329.

[11] F. ROSENBLATT, "The perceptron: a probabilistic model for information storage and organization in the brain", *Psychol Rev*, vol. 65, no. 6, pp. 386–408, Nov. 1958.

[12] Z. Peng, *Multilayer perceptron algebra*, 2017. arXiv: 1701.04968 [stat.ML].

[13] S. Jiang and V. M. Zavala, "Convolutional neural nets in chemical engineering: Foundations, computations, and applications", *AIChE Journal*, vol. 67, no. 9, May 2021. DOI: 10.1002/aic.17282. [Online]. Available: https://doi.org/10.1002%5C%2Faic.17282.

[14] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", *Biological Cybernetics*, 1980. DOI: 10.1007/BF.1980.00344251.

[15] A. Jain, H. Patel, L. Nagalapatti, *et al.*, "Overview and importance of data quality for machine learning tasks", in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 3561–3562, ISBN: 9781450379984. DOI: 10.1145/3394486.3406477. [Online]. Available: https://doi.org/10.1145/3394486.3406477.

[16] E. D. Cristofaro, *An overview of privacy in machine learning*, 2020. arXiv: 2005.08679 [cs.LG].

[17] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world", *CoRR*, vol. abs/1607.02533, 2016. arXiv: 1607.02533. [Online]. Available: http://arxiv.org/abs/1607.02533.

[18] X. Ying, "An overview of overfitting and its solutions", *Journal of Physics: Conference Series*, vol. 1168, no. 2, p. 022 022, Feb. 2019. DOI: 10.1088/1742-6596/1168/2/022022. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1168/2/022022.

[19] P. Kairouz, H. B. McMahan, B. Avent, *et al.*, "Advances and open problems in federated learning", *CoRR*, vol. abs/1912.04977, 2019. arXiv: 1912.04977. [Online]. Available: http://arxiv.org/abs/1912.04977.

[20] A. Salama, A. Stergioulis, A. M. Hayajneh, S. A. R. Zaidi, D. McLernon, and I. Robertson, "Decentralized federated learning over slotted aloha wireless mesh networking", *IEEE Access*, vol. 11, pp. 18 326–18 342, 2023. DOI: 10.1109/ACCESS.2023.3246924.

[21] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, *Communication-efficient learning of deep networks from decentralized data*, 2023. arXiv: 1602.05629 [cs.LG].

[22] B. McMahan and D. Ramage, *Federated learning: Collaborative machine learning without centralized training data*, Apr. 2017. [Online]. Available: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[23] Y. Hua, K. Miller, A. L. Bertozzi, C. Qian, and B. Wang, *Efficient and reliable overlay networks for decentralized federated learning*, 2021. arXiv: 2112.15486 [cs.NI].

[24] S. P. Karimireddy, M. Jaggi, S. Kale, *et al.*, "Breaking the centralized barrier for cross-device federated learning", in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, Curran Associates, Inc., 2021, pp. 28 663–28 676. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/f0e6be4ce76ccfa73c5a540d992d0756-Paper.pdf.

[25] P. M. S. Sánchez, A. H. Celdrán, E. T. M. Beltrán, *et al.*, *Analyzing the robustness of decentralized horizontal and vertical federated learning architectures in a non-iid scenario*, 2022. arXiv: 2210.11061 [cs.LG].

[26] M. Yemini, R. Saha, E. Ozfatura, D. Gündüz, and A. J. Goldsmith, *Semi-decentralized federated learning with collaborative relaying*, 2022. arXiv: 2205.10998 [cs.LG].

[27] A. Bellet, A. Kermarrec, and E. Lavoie, "D-cliques: Compensating noniidness in decentralized federated learning with topology", *CoRR*, vol. abs/2104.07365, 2021. arXiv: 2104.07365. [Online]. Available: https://arxiv.org/abs/2104.07365.

[28] L. Huang, A. L. Shea, H. Qian, A. Masurkar, H. Deng, and D. Liu, "Patient clustering improves efficiency of federated machine learning to predict mortality and hospital stay time using distributed electronic medical records", en, *J. Biomed. Inform.*, vol. 99, no. 103291, p. 103 291, Nov. 2019.

[29] S. Lu, Y. Zhang, and Y. Wang, "Decentralized federated learning for electronic health records", in *2020 54th Annual Conference on Information Sciences and Systems (CISS)*, 2020, pp. 1–5. DOI: 10.1109/CISS48834.2020.1570617414.

[30] B. Ikiz, *A pandemic ai engine without borders*, Aug. 2020. [Online]. Available: https://hai.stanford.edu/news/pandemic-ai-engine-without-borders.

[31] S. Savazzi, M. Nicoli, M. Bennis, S. Kianoush, and L. Barbieri, "Opportunities of federated learning in connected, cooperative, and automated industrial systems", *IEEE Communications Magazine*, vol. 59, no. 2, pp. 16–21, 2021. DOI: 10.1109/MCOM.001.2000200.

[32] Y. Qu, S. R. Pokhrel, S. Garg, L. Gao, and Y. Xiang, "A blockchained federated learning framework for cognitive computing in industry 4.0 networks", *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2964–2973, 2021. DOI: 10.1109/TII.2020.3007817.

[33] H. Wang, L. Muñoz-González, D. Eklund, and S. Raza, "Non-iid data re-balancing at iot edge with peer-to-peer federated learning for anomaly detection", in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '21, Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2021, pp. 153–163, ISBN: 9781450383493. DOI: 10.1145/3448300.3467827. [Online]. Available: https://doi.org/10.1145/3448300.3467827.

[34] J. Kang, D. Ye, J. Nie, *et al.*, *Blockchain-based federated learning for industrial metaverses: Incentive scheme with optimal aoi*, 2022. arXiv: 2206.07384 [cs.GT].

[35] M. Paulik, M. Seigel, H. Mason, *et al.*, *Federated evaluation and tuning for on-device personalization: System design & applications*, 2021. arXiv: 2102.08503 [cs.LG].

[36] Y. Belal, A. Bellet, S. B. Mokhtar, and V. Nitu, "PEPPER", *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 6, no. 3, pp. 1–27, Sep. 2022. DOI: 10.1145/3550302. [Online]. Available: https://doi.org/10.1145%5C%2F3550302.

[37] M. Khelghatdoust and M. Mahdavi, "A socially-aware, privacy-preserving, and scalable federated learning protocol for distributed online social networks", in *Advanced Information Networking and Applications : Proceedings of the 36th International Conference on Advanced Information Networking and Applications (AINA-2022), Volume 2*, ser. Lecture Notes in Networks and Systems, 2022, pp. 192–203, ISBN: 978-3-030-99587-4. DOI: 10.1007/978-3-030-99587-4_17.

[38] T. Wang, Y. Liu, X. Zheng, H.-N. Dai, W. Jia, and M. Xie, "Edge-based communication optimization for distributed federated learning", English, *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 4, pp. 2015–2024, Jul. 2022, ISSN: 2327-4697. DOI: 10.1109/TNSE.2021.3083263.

[39] J. Ding, E. Tramel, A. K. Sahu, S. Wu, S. Avestimehr, and T. Zhang, *Federated learning challenges and opportunities: An outlook*, 2022. arXiv: 2202.00807 [cs.LG].

[40] X. Yu, L. Li, X. He, S. Chen, and L. Jiang, "Federated learning optimization algorithm for automatic weight optimal", *Computational Intelligence and Neuroscience*, vol. 2022, 2022. DOI: https://doi.org/10.1155/2022/8342638. [Online]. Available: https://doi.org/10.1155/2022/8342638.

[41] Y. Huang and C. Hu, *Toward data heterogeneity of federated learning*, 2022. arXiv: 2212.08944 [cs.LG].

[42] T. Orekondy, S. J. Oh, Y. Zhang, B. Schiele, and M. Fritz, *Gradient-leaks: Understanding and controlling deanonymization in federated learning*, 2020. arXiv: 1805.05838 [cs.CR].

[43] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions", *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, May 2020. DOI: 10.1109/msp.2020.2975749. [Online]. Available: https://doi.org/10.1109%5C%2Fmsp.2020.2975749.

[44] E. T. M. Beltrán, *Fedstellar: Framework for decentralized federated learning*, 2023. [Online]. Available: https://federatedlearning.inf.um.es/.

[45] Y. LeCun, C. Cortes, and C. Burges, *The mnist database*, 1994. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[46] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "EMNIST: an extension of MNIST to handwritten letters", *CoRR*, vol. abs/1702.05373, 2017. arXiv: 1702.05373. [Online]. Available: http://arxiv.org/abs/1702.05373.

[47] S. Caldas, P. Wu, T. Li, *et al.*, "LEAF: A benchmark for federated settings", *CoRR*, vol. abs/1812.01097, 2018. arXiv: 1812.01097. [Online]. Available: http://arxiv.org/abs/1812.01097.

[48] J. N. Kather, C.-A. Weis, F. Bianconi, *et al.*, "Multi-class texture analysis in colorectal cancer histology", *Scientific Reports*, vol. 6, no. 1, 2016. DOI: 10.1038/srep27988.

[49] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images", University of Toronto, Toronto, Ontario, Tech. Rep. 0, 2009.

[50] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms", *CoRR*, vol. abs/1708.07747, 2017. arXiv: 1708.07747. [Online]. Available: http://arxiv.org/abs/1708.07747.

[51] V. Shejwalkar, A. Houmansadr, P. Kairouz, and D. Ramage, "Back to the drawing board: A critical evaluation of poisoning attacks on federated learning", *CoRR*, vol. abs/2108.10241, 2021. arXiv: 2108.10241. [Online]. Available: https://arxiv.org/abs/2108.10241.

[52] M. Fang, X. Cao, J. Jia, and N. Z. Gong, "Local model poisoning attacks to byzantine-robust federated learning", *CoRR*, vol. abs/1911.11815, 2019. arXiv: 1911.11815. [Online]. Available: http://arxiv.org/abs/1911.11815.

[53] V. Tolpegin, S. Truex, M. Emre Gursoy, and L. Liu, "Data Poisoning Attacks Against Federated Learning Systems", *arXiv e-prints*, arXiv:2007.08432, arXiv:2007.08432, Jul. 2020. arXiv: 2007.08432 [cs.LG].

[54] M. Baruch, G. Baruch, and Y. Goldberg, "A little is enough: Circumventing defenses for distributed learning", *CoRR*, vol. abs/1902.06156, 2019. arXiv: 1902.06156. [Online]. Available: http://arxiv.org/abs/1902.06156.

[55] V. Shejwalkar and A. Houmansadr, "Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning", in *NDSS*, 2021.

[56] C. Xie, O. Koyejo, and I. Gupta, "Generalized byzantine-tolerant SGD", *CoRR*, vol. abs/1802.10116, 2018. arXiv: 1802.10116. [Online]. Available: http://arxiv.org/abs/1802.10116.

[57] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. B. Calo, "Analyzing federated learning through an adversarial lens", *CoRR*, vol. abs/1811.12470, 2018. arXiv: 1811.12470. [Online]. Available: http://arxiv.org/abs/1811.12470.

[58] V. Tolpegin, S. Truex, M. Emre Gursoy, and L. Liu, "Data Poisoning Attacks Against Federated Learning Systems", *arXiv e-prints*, arXiv:2007.08432, arXiv:2007.08432, Jul. 2020. arXiv: 2007.08432 [cs.LG].

[59] E. M. E. Mhamdi, R. Guerraoui, and S. Rouault, *The hidden vulnerability of distributed learning in byzantium*, 2018. DOI: 10.48550/ARXIV.1802.07927. [Online]. Available: https://arxiv.org/abs/1802.07927.

[60] M. Adnan, S. Kalra, J. C. Cresswell, G. W. Taylor, and H. Tizhoosh, "Federated learning and differential privacy for medical image analysis", *Scientific Reports*, 2021. DOI: 10.21203/rs.3.rs-1005694/v1.

[61] P. Blanchard, E. M. E. Mhamdi, R. Guerraoui, and J. Stainer, "Byzantine-tolerant machine learning", *CoRR*, vol. abs/1703.02757, 2017. arXiv: 1703.02757. [Online]. Available: http://arxiv.org/abs/1703.02757.

[62] A. Trask, D. Gilmore, and M. Russell, "Modeling order in neural word embeddings at scale", *CoRR*, vol. abs/1506.02338, 2015. arXiv: 1506.02338. [Online]. Available: http://arxiv.org/abs/1506.02338.

[63] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent", in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/f4b9ec30ad9f68f89b29639786cb62ef-Paper.pdf.

[64] D. Yin, Y. Chen, K. Ramchandran, and P. L. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates", *CoRR*, vol. abs/1803.01498, 2018. arXiv: 1803.01498. [Online]. Available: http://arxiv.org/abs/1803.01498.

[65] E. M. E. Mhamdi, R. Guerraoui, and S. Rouault, *The hidden vulnerability of distributed learning in byzantium*, 2018. arXiv: 1802.07927 [stat.ML].

[66] *GitHub - tensorflow/tensorboard: TensorFlow's Visualization Toolkit — github.com*, https://github.com/tensorflow/tensorboard, [Accessed 25-May-2023].

[67] L. Muñoz-González, B. Biggio, A. Demontis, *et al.*, "Towards poisoning of deep learning algorithms with back-gradient optimization", *CoRR*, vol. abs/1708.08689, 2017. arXiv: 1708.08689. [Online]. Available: http://arxiv.org/abs/1708.08689.

[68] S. Sengupta, A. Dudley, T. Chakraborti, and S. Kambhampati, "An investigation of bounded misclassification for operational security of deep neural networks", 2018.

[69] G. Rodola, *Psutil documentation*, 2023. [Online]. Available: https://psutil.readthedocs.io/en/latest/index.html.

Abbreviations

ANN	Artificial Neural Network
AGR	Aggregation Rule
API	Application Programming Interface
ASR	Automatic Speech Recognition
CFL	Centralized Federated Learning
CNN	Convolutional Neural Network
CT	Computed Tomography
CV	Computer Vision
D2D	Device to Device
DFL	Decentralized Federated Learning
EDA	Exploratory Data Analysis
FL	Federated Learning
FNN	Feedforward Neural Network
IoT	Internet of Things
IID	Independent and identically distributed
ML	Machine Learning
MLP	Multilayer Perceptron
SDFL	Semi-Decentralized Federated Learning
SGD	Stochastic Gradient Descent

List of Figures

List of Tables

Listings

Appendix A

Installation Guidelines

1. Prerequisites:
   - Python version 3.10 [1]
   - pip3
2. Unpack the ”fedstellar-modified.zip”-File
3. Access the directory using any terminal and create log and model destination

   ```
   $ cd fedstellar-modified
   $ mkdir -p app/logs && mkdir -p app/models
   ```
4. Create and activate virtual environment (venv):

   ```
   $ python3 -m venv fedstellar-venv
   ```
   If you use bash, activate the venv as follows:

   ```
   $ . fedstellar-venv/Scripts/activate
   ```
   If you use zsh/csh/tcsh, activate the venv as follows:

   ```
   $ source fedstellar-venv/Scripts/activate
   ```
5. Install Python requirements

   ```
   $ pip3 install -r requirements.txt
   ```
6. Run FedStellar

   ```
   $ python app/main.py --webserver --python <python-path>
   ```
   replace <path> with the absolute path to the python executable.

The FedStellar Webserver should now be available on http://127.0.0.1:5000. Sign in using the username ”admin” with the password ”admin” - to use the implementations of this work, go to ”Scenario Management” and then ”Deploy Scenario”.

[1] Other Python versions >3.8 should be compatible, but have not been tested.

Appendix B

Contents of the .zip-File

- BA-timothy-naescher.pdf - Final Report
- BA-timothy-naescher.zip - LaTeX source code of Final Report.
- fedstellar-ba.zip - FedStellar Framework including all implementations
- evaluation.zip - Graphs and raw data of all evaluations
- midterm-slides.pptx - Midterm Presentation Slides
- final-slides.pptx - Final Presentation Slides

Appendix C

Model Summaries

Table C.1: CIFAR10: MLP

CIFAR-10: MLP				
	Name	Type	Output Shape	Params
0	metric	MulticlassAccuracy	[1, 10]	
1	l1	Linear	[1, 512]	1'573'376
2	l2	Linear	[1, 256]	131'328
3	l3	Linear	[1, 10]	2'570

1.7M (1.7M) Total (Trainable) params
6.85 MB Model params size (estimate)
Optimizer: Adam

Table C.2: MNIST: MLP

MNIST: MLP				
	Name	Type	Output Shape	Params
0	metric	MulticlassAccuracy	[1, 10]	
1	l1	Linear	[1, 256]	200'960
2	l2	Linear	[1, 128]	32'896
3	l3	Linear	[1, 10]	1'290

235 K (235 K) Total (Trainable) params
0.95 MB Model params size (estimate)
Optimizer: Adam

Table C.3: Fashion-MNIST: MLP

Fashion-MNIST: MLP				
	Name	Type	Output Shape	Params
0	metric	MulticlassAccuracy	[1, 10]	
1	l1	Linear	[1, 256]	200'960
2	l2	Linear	[1, 128]	32'896
3	l3	Linear	[1, 10]	1'290

235 K (235 K) Total (Trainable) params
0.95 MB Model params size (estimate)
Optimizer: Adam