Eduard Josep Bel Ribes

LEVERAGING INTER- AND INTRA-CLASS DISTANCES FOR POISONING ATTACKS

MASTER'S THESIS

Directed by Dr. Alberto Blanco Justicia

Master's Degree in Computer Security Engineering and Artificial Intelligence

UNIVERSITAT ROVIRA i VIRGILI

Tarragona
2023

Acknowledgements

I want to thank...
blablabla

Resum

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat.

Resumen

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna.

Abstract

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna.

Contents

List of code snippets

List of Figures

List of Tables

1 Introduction

In today's age of information and connectivity, advances in Artificial Intelligence (AI) and Machine Learning (ML) have transformed the way users interact with technology and process data.

Among the emerging paradigms in the field of ML, Federated Learning (FL) appeared as an innovative approach to train AI models in a distributed and decentralized environment.

In the last decade, ML has revolutionized the way in which we face complex problems in different areas, from computer vision to natural language processing. The last two years have been filled with news about promissing new paradigms of image generation, classification, chatbots, speech regonition and other AI's and, as time goes by, the applications of AI are becoming more present in our daily lives.

We can find examples of applications using FL in examples such as the text predictive keyboard that we can find on our mobile phones (Google's Android Keyboard [1]). We also find FL in Apple's assistant Siri voice recognition. This technology helps distinguish wether it is the main user of the smartphone saying "Hey, Siri", or another iPhone user attempting to activate Siri on their phone. As a final example, FL is used in more complex applications such as the Tesla autonomous driving system. In all three cases, the use of FL allows the machine learning models to be trained with the users' data without them having to share their data with third parties. In the case of Google's predictive keyboard, the model is trained with the users' data locally on the device, while in the case of Tesla, the users' data is used to train the model in a distributed way among the vehicles in the Tesla fleet.

Despite its advantages in terms of privacy, scalability and efficiency, FL presents significant challenges. One of them is its vulnerability to attacks: these attacks can exploit the distributed nature of the learning process, aiming to compromise integrity, confidentiality and the model's efficiency. As seen in the real-world examples, if an attacker exploited a vulnerability of the Tesla autonomous driving system by modifying the action that the car takes when recognizing a STOP sign, changing it to accelerating at full capacity, it could lead to multiple car accidents.

As Federated Learning systems are increasingly integrated in real-world applications, it becomes necessary to understand and address these challenges to guarantee a successful and secure deployment of this technology.

The GitHub repository "LFighter" [2] by Najeeb Jabreel is this thesis inspiration. It already offers a working FL simulator, where the programmer can modify the environment's parameters and obtain results of attacks against a variety of servers with different rules. With the explicit purpose of formulating and deploying attacks against FL systems to assess their robustness, this simulator serves as the foundation for the development of this thesis.

1.1 Motivation

Since the paper proposing FL got published in 2017, this new paradigm has been increasingly used over the years as we can see in Figure 1. With its increase in popularity and, its implementation in sensitive applications such as Tesla's autonomous driving system, our concerns about the security of this technology also increase.

From this concern, the motivation of detecting possible vulnerabilities so they can be addressed arises.

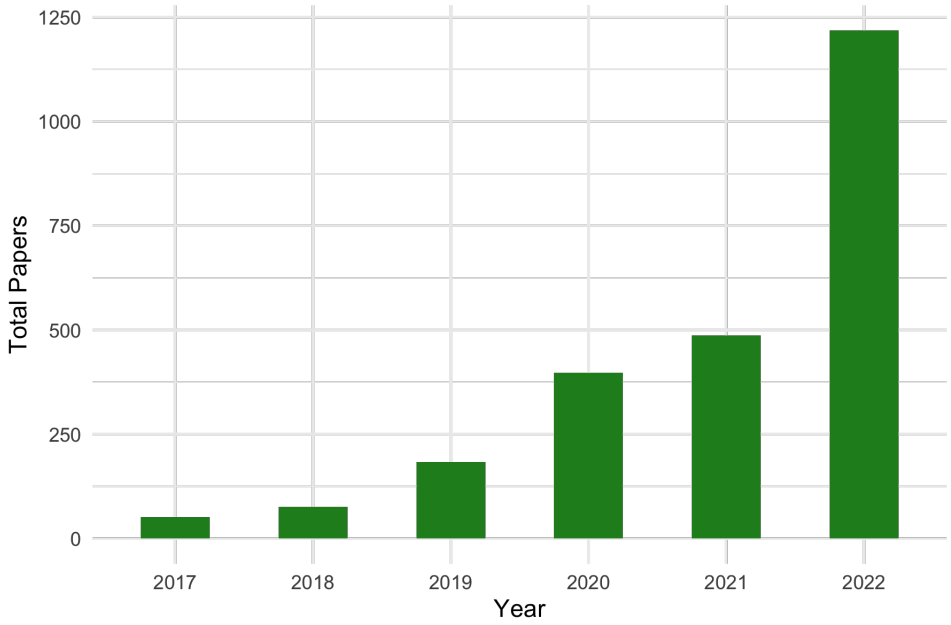

Figure 1: Published papers per year since FL was proposed. Source: *Web of Science*

1.2 Objectives

The main theoretical objective of this thesis is to explore and examine in detail the kind of attacks that can be directed towards Federated Learning systems, as well as identifying strategies and solutions to mitigate these attacks.

The practical objective is to examine the effectiveness and implications of employing a sophisticated label flipping technique compared to a straightforward approach when targeting a Federated Learning system. Specifically, the focus is on evaluating whether a more strategic and intelligent choice of samples to suffer a label flipping can lead to greater success for potential attackers compared to indiscriminately flipping all labels. This investigation represents an essential step towards comprehending the vulnerabilities and potential weak points within the Federated Learning paradigm. Furthermore, we aim to provide results on some of the most used aggregation rules in Federated Learning systems.

The research may be able to identify patterns and insights that conventional methods of attack might miss by examining the results of strategically manipulated label flipping and comparing them with the brute-force method. The results of this thesis will be valuable in gaining a deeper understanding of the security implications of FL and to develop more robust and secure FL systems in case the conceived attacks succeed.

In summary, this thesis conducts a critical examination of the viability of using intelligent label flipping techniques in comparison to a brute-force approach when attacking a Federated Learning system.

1.3 Outline

This thesis is organized as follows: . IDJQWOIJIOQ

. DHUIQWIHDIUQWUHI

2 Background

In this section, we establish the essential groundwork for comprehending the intricacies of model poisoning through label flipping on FL. We begin by presenting important ideas required to understand its security landscape.

We examine the state of the art with regard to adversarial attacks and defence mechanisms to get a practical perspective on security challenges. By exploring the landscape of model poisoning attacks, we will delve into attacker tactics that manipulate model updates, potentially compromising the integrity of federated learning

Finally, we explore the different aggregation techniques that are going to be employed on the practical side of this thesis.

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a specific type of Machine Learnig (ML) algorithms characterized by their incorporation of multiple hidden layers situated between the input and output layers. These hidden layers allow DNNs to acquire intricate and refined data representations during training. This inherent capability enhances their performance across diverse domains, encompassing healthcare [3], game playing, speech recognition, analysing molecular structures and predicting chemical properties, recommendation systems, and many others.

The DNN first creates a map of virtual neurons and, assigns to the connections linking them initial random "weights", or numerical values. These weights interact with the inputs to produce an output that can range from 0 to 1. An algorithm intervenes to adjust these weights if the network is unable to recognise a particular pattern with sufficient accuracy. Through this procedure, the algorithm can gradually increase the influence of specific parameters while iteratively determining the most effective mathematical operations necessary to efficiently process the input data. Figure 2 shows a representation of a DNN that classifies images into classes of animals.

For instance, a DNN trained to recognise plant species will examine the supplied image and determine whether the plant in the image is a member of a particular species. The user can then evaluate the outcomes and select the probabilities that the network should present (those that are higher than a certain threshold, etc.), resulting in the suggested label.

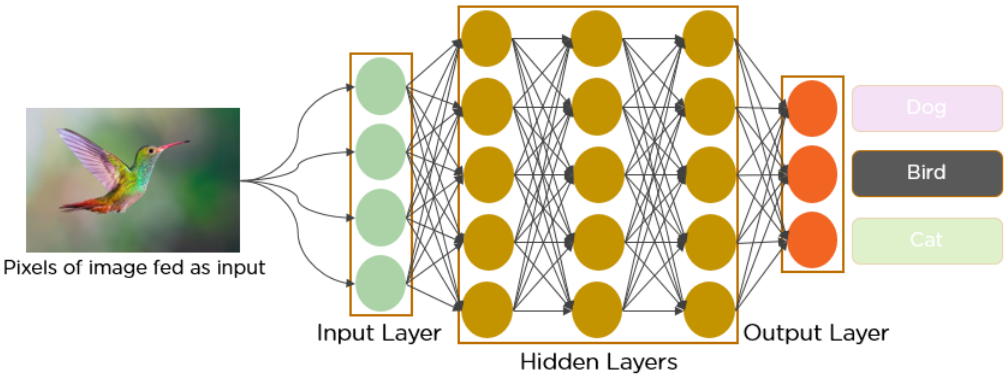

Figure 2: Deep Neural Network representation. Source: *Analytics Vidhya*

2.2 Federated Learning

Federated Learning, first proposed by McMahan et al. in 2017 [4], has become recognised as an innovative paradigm in the context of ML. In the age of distributed computing and data privacy concerns, FL offers an innovative approach of model training. By allowing ML models to be trained collaboratively across decentralised devices while protecting the confidentiality of specific data sources, it addresses the issue of centralised data ownership and the privacy implications it has.

The FL training procedure is an excellent representation of a distributed and privacy-protecting mechanism. Using their private data, participating devices or clients refine specific models in this scheme. The global model is used as the starting point for each client's training process and is initialised and distributed by a central server. Clients produce incremental model updates by iteratively optimising this global model through their local datasets. The central server receives these updates, compiles them, and distributes the new global model again to the clients. This method's elegance lies in its capacity to combine information from various data sources while protecting private data at the source.

Due to its collaborative and iterative structure, FL has special qualities that set it apart from conventional centralised learning paradigms. The training dynamics are made more complex by the presence of device-specific data distributions and a variety of client computational capabilities. Therefore, FL encompasses challenges that go beyond those of conventional ML, calling for the investigation of reliable communication protocols, secure aggregation techniques, and methods for dealing with potential adversarial threats.

Compared to conventional centralised ML approaches, FL offers a number of compelling advantages:

- The training computational load is effectively distributed across the participating peers' devices, which is particularly important for large-scale ML tasks.
- Joint training on various data sources improves model accuracy and yields more accurate insights for both peers and the central server.
- By eliminating the requirement to share local data with a central server, FL crucially protects individual privacy.

Due to the latter advantage, FL is particularly suitable for scenarios involving sensitive data, such as those in location-based services, voice assistants, healthcare, facial recognition, and voice assistants. Additionally, FL is extremely useful in scenarios where data processing and collection are limited by privacy protection laws like the General Data Protection Regulation (GDPR [5]) by the European Commission or the Spanish "*Ley Orgánica de Protección de Datos*" (LOPD).

2.3 State of the art

Despite the numerous advantages that FL offers over centralised learning, the decentralised nature of this approach also creates vulnerabilities to security and privacy threats. In fact, the distributed architecture that empowers FL, can increase the impact of these attacks, surpassing the risks associated with more conventional centralised learning.

Numerous security and privacy issues can still affect FL. On the security front, the vulnerabilities include Byzantine attacks that aim to obstruct model convergence and, poisoning attacks, that are deliberately designed to influence convergence in the wrong direction.

2.3.1 Privacy attacks

While FL works to prevent direct sharing of private data, the process of exchanging local updates creates the possibility of sensitive information leakage to malicious actors.

The gradients computed on individual devices have the potential to unintentionally make adversaries aware of subtle aspects of the training data. Deep Learning models have an extraordinary capacity to retain information beyond what is strictly required for their primary task, and this tendency can unintentionally reveal unintended properties of the data they were trained on.

Peers' local updates reflect the understanding gained from their different training datasets. As a result, these updates have the unintentional potential to reveal personal information, such as class distributions, membership details, and inherent properties of the local training data. This unintentionally gives adversaries the ability to infer class labels by reconstructing training samples without having any prior knowledge of the underlying data.

2.3.2 Poisoning attacks against Federated Learning

FL is vulnerable to poisoning attacks, a technique designed to sabotage the learning process by introducing malicious data or model updates. These attacks within FL systems can be divided into two categories: untargeted and targeted.

Understanding the landscape of poisoning attacks against FL is crucial to safeguarding the integrity of the learning process. During FL's training phase, both targeted and untargeted poisoning attacks can be executed, influencing either the local model or the local data. The act of injecting manipulated samples into the training dataset is known as "data poisoning attack," capable of introducing distortions by feeding the model with inaccurate or biased data. In contrast, model poisoning attacks involve the manipulation of model parameters during the local model training, either directly or indirectly.

2.3.3 Untargeted poisoning attacks

Untargeted poisoning attacks in the context of FL focus on degrading the model's overall performance rather than aiming for particular misclassifications. Without following a predetermined pattern, these attacks introduce noise or perturbations into the training process. The result is a compromised global model with decreased prediction accuracy and reliability.

Byzantine attacks are a subset of untargeted attacks that involve malicious devices deliberately sending false updates to the central server during the model aggregation phase. In order to prevent the convergence of the global model from happening, these malicious devices act dishonestly by transmitting updated model data that has been altered or corrupted. Unaware of the adversarial behaviour, the central server aggregates these updates, creating a distorted model that does not accurately reflect the underlying data.

Due to the hidden nature of the malicious actions, which can closely resemble legitimate participation, detecting Byzantine attacks can be challenging. Standard aggregation

techniques might unintentionally include these contaminated updates, which would make the global model perform poorly on unobserved data.

2.3.4 Targeted poisoning attacks

Targeted poisoning attacks have a specific goal: inducing the global model to misclassify a chosen set of samples into a target class chosen by the attacker. The specific targeted attack that holds relevance for this thesis is the label-flipping attack.

In a label-flipping attack, attackers employ their local dataset to carry out their poisoning strategy in the following way: for each instance in the dataset that originally has the source class label, they meticulously change the label to the target one. These attackers then proceed to train their local models after manipulating their training data. The parameters used in this training process are the same as those provided by the central server. As a result, the model learns on these incorrectly labelled examples, which causes it to produce inaccurate predictions in the future when presented with images of a similar nature.

Consider a scenario for a medical diagnosis where a FL model is trained to distinguish between samples that are healthy and those that are diseased. A label-flipping attack could be carried out by an attacker by changing the labels of some healthy samples to read "diseased." The model then gains knowledge from these falsified data, misclassifying real healthy samples as diseased during inference. In real-world applications, such as incorrect diagnoses or treatment recommendations, such a scenario might have serious consequences.

2.3.5 Defenses against poisoning attacks

The way for a malicious actor to compromise the integrity of the learning process is, as seen in the previous sections 2.3.3 and 2.3.4, by injecting poisoned data on local model updates. The attacker must also overcome the combined influence of benign clients during the aggregation process in order ensure a successful attack. This can be done in a number of ways, including:

- Applying scaling factors to boost the impact of their own updates.
- Colluding with other malicious clients, whether they are additional accounts linked to the same attacker or different attackers themselves.
- Applying a combination of the aforementioned strategies.

A useful strategy to prevent attacks of this nature would be, given that the server possesses the some unique identifier (ID) of peers participating in the current training round, contrasting the model's accuracy results with those of prior rounds. In rounds of noteworthy disparities, the algorithm could store the IDs of participants from that specific round. In subsequent rounds marked by discrepancies, this stored list could be cross-referenced with the current round IDs, allowing for the removal of IDs not actively participating in the ongoing round. This strategy would eventually facilitate the detection of a single malicious peer. Notably, this identification does not require to ban the detected malicious peer, as this might lead her to create a new account to evade detection. For scenarios involving multiple malicious peers, an iterative process would take place, persisting until the identification of additional attackers is achieved. To establish a trustable first global round for future

comparisons, the server could initiate this round with the assurance of a lack of attackers at the chosen peers set.

A less 'drastic' approach to blacklisting is using reputations, rewarding good updates and penalizing bad ones, and during aggregation, weighting updates based on peers' reputations. Thus, updates from suspicious peers carry less weight than those from trustworthy ones. The issue with this mechanism is that it assumes the central server has access to sufficient testing data for model evaluation. In a FL setting, this doesn't always hold true, which is why the mechanisms discussed below have been proposed.

The strategies for defending against poisoning attacks that have been suggested in the literature follow one of the following principles:

- Update Aggregation: In this method, local model updates are aggregated using methods that are robust to outliers. The impact of inaccurate updates on the final global model is reduced by using aggregation techniques that are resilient to extreme values. This ensures that the effects of potentially harmful updates are minimised, allowing to produce an aggregated model that is more reliable and accurate.
- Evaluation Metrics: The central idea of this approach is to evaluate the quality of local updates using evaluation metrics connected to the global model. The model aggregation process may exclude or penalise a local update if it negatively impacts a certain metric, such as accuracy.
- Update Clustering: In a different approach, updates are split into two clusters, with the smaller cluster being marked as potentially malicious and ignored during model learning. This idea, helps filter out potentially harmful updates.
- Peers' Behaviour: This approach makes the assumption that malicious peers behave similarly, which makes their updates more similar than those of honest peers. In order to reduce the impact of potentially harmful updates, penalization is therefore based on the similarity between updates.
- Differential Privacy (DP): Using the DP method, each update parameter is altered by being clipped to a maximum threshold and then introducing random noise. As a result, there is a compromise between the ability of the aggregated model to perform its main task and the mitigation of potential attacks through added noise.

The strategies implemented in the base code that is used in this thesis [2] are:

- Federated Averaging (FedAvg) [4]: The standard aggregation method used in FL. It works by aggregating the local updates by averaging them. This method is not designed to counter poisoning attacks.
- Median [6]: This strategy involves collecting local model updates from participating devices and determining the median value for each parameter across these updates. After sorting the parameter values, the median strategy selects the middle value while ignoring extreme outliers. Since a single adversarial device cannot significantly impact the final aggregated model, this method makes the median aggregation resistant to malicious or inaccurate updates.

- Trimmed Mean (TMean) [6]: This method decreases the impact of outliers by excluding a certain percentage of extreme values. By reducing the impact of potentially malicious updates or noisy data from individual devices, this strategy improves the robustness of the aggregation process.

- Multi-Krum (MKrum) [7]: The updates chosen for aggregation using this method are the most agreeable ones. The algorithm isolates potentially malicious or abnormal updates by selecting a subset of updates with the highest consensus among participating devices. The MKrum strategy improves the robustness of aggregation against adversarial behaviour and data anomalies by focusing on the level of agreement among multiple devices.

- FoolsGold (FGold) [8]: By taking into account the similarity of their contributions, the method adapts the learning rate of clients. The central idea of this strategy is based on the notion that when a group of sybils[1] manipulates a shared model, their updates over the course of training will align towards a particular malicious objective, displaying a higher degree of similarity than expected. In FL, it serves as a robust defence mechanism against assaults planned by any number of sybils.

- Tolpegin [9]: The weights associated with the potentially targeted source class are examined by this method using Principal Component Analysis (PCA)[2]. Within those weight distributions, it selectively eliminates potential adversarial updates that deviate from the prevailing trend.

- FLAME [10]: Model clustering and weight clipping are two techniques FLAME employs to reduce the required noise infusion. This method accomplishes two goals through this method: successfully closing any potential adversarial backdoors while maintaining the aggregate model's desirable performance.

- LFighter [11]: This method effectively filters out updates that might be harmful before model aggregation by extracting gradients corresponding to potential source and target classes from local updates, clustering them. It's noteworthy that this proposed defence is robust across various data distributions and model dimensions.

[1] A term used in computer security to describe a scenario in which a user or entity generates several different entities.

[2] PCA is a statistical technique used to reduce the dimensionality of data while preserving its essential features.

3 Architecture

In this section dedicated to the architecture of the thesis, we will delve into the organizational framework of the foundational code derived from the GitHub repository [2]. We will discuss the structural elements that constitute the basis of our work, elucidating the positioning of the newly crafted code that serves the thesis purpose. The goal is to demonstrate how the intricate nature of a typical FL environment is mirrored in the architecture of this implementation by drawing comparisons to real-world FL scenarios and the base code structure.

3.1 Base code structure

The base code is implemented in Python and structured as follows:

- Python notebooks (.ipynb files): There are three notebooks, one for each dataset used in the developer's experiments. The included datasets are:
  - MNIST [12]: This dataset contains samples of handwritten digits. It consists of a training set of 60,000 samples, and a test set of 10,000 samples. Each sample is a 28x28 bit grayscale image, associated with a label from 10 classes. The task is to classify the images into their respective digit classes.
  - CIFAR-10 [13]: This dataset contains 60,000 32x32 bit color images in 10 different classes. These classes encompass a diverse array of objects, including but not limited to animals, vehicles, and everyday items. The objective underlying this dataset is to accurately classify each image into its designated class.
  - IMDB [14]: This dataset contains 50,000 movie reviews from the Internet Movie Database. The task is to classify the reviews into positive or negative sentiment.

  The notebooks are used to run the experiments using the different aggregation functions commented in section 2.3.5 at user's will. This is done by importing the libraries and defining global variables in one executable section, and arranging the tests with different aggregation methods into separate sections. These notebooks are also used to visualize the results and checking the process.

- *experiment_federated.py*: This python file contains a sole function (*run_exp()*) that is called by one of the Python notebooks whenever we wish to start a new experiment with an aggregation function. This function merely prints by console information concerning the parameters used in the current experiment, initializes the FL environment and, calls a more complex function.

- *environment_federated.py*: This is the most complex file in the base code. It contains the *run_experiment()* function, which is the one called by the previous file. This function is responsible for the execution of the experiment, and it is where the whole FL setting is initialized and simulated. The file is divided into two main classes:
  - Peer: When a Peer object is created, the Peer class initializes all its variables, such as the peer's ID, its local data, etc. This class contains a single function, named *participant_update()*, which is called when a peer has to update its local model. This function is responsible for the training of the local model, and it is

where, depending on the value of the parameter "*attack_type*", the execution of the attack is determined. The function returns the updated local model.

- FL: This class is responsible for the initialization of the FL environment, from the most simple parameters such as the global rounds, to the more complex tasks such as creating the peers' instances and setting the global model up. It contains four functions:
  - *test()*: This function is used to test the model's accuracy.
  - *test_label_predictions()*: As the name suggests, this function is used to test the label predictions of the model received as a parameter. It is responsible for returning a list with the actual labels and another with the predicted ones in order to obtain the accuracy of the model.
  - *choose_peers()*: This function selects *n* random peers from the list of peers. The value of *n* is determined by the total amount of peers and the malicious rate, which is a floating-point variable.
  - *run_experiment()*: This is the function that simulates the whole FL environment. The function is built as follows:
    1. It begins by copying the global model into a local variable.
    2. After that, it iterates through a loop a total of "*global_rounds*" times. Inside this loop, for each global round, it selects the peers that will participate in the current round by calling the *choose_peers()* function, it also reinitializes the utility tables (weights, local models, etc.) of the peers that will participate in the current round, and then, for each peer participating:
       (a) It defines the peer as attacker or regular depending on the output of *choose_peers()*, calls the *participant_update()* function of the peer.
       (b) It updates the utility tables of the peer with the new values.
    3. After all peers have been updated, it aggregates the local models of the peers that participated in the current round by the selection of the aggregation function dependant on a series of conditional statements.
    4. It updates the global model with the aggregated model and the process is repeated until the global rounds are completed.
    5. The *test()* function is called to obtain the model's accuracy.
    6. Finally, it returns the system's state by console.

The logical location for the newly created code, explained in Section 4 is inside the *environment_federated.py* file. The exact placement for these functions is inside the *participant_update()* function, contained by the Peer class. This is because it is the single point of entry for the execution of the attack, and it is where the local model is updated.

3.2 Real-world FL vs. base code

In this section, we state the similarities between a real FL system and the base code described in the previous section, 3.1. As a navigational aid as we navigate through the details of these two distinct environments, Figure 3 provides a diagram of an actual FL scenario to further illuminate this comparison.

Next, the steps comprising the diagram are briefly outlined:

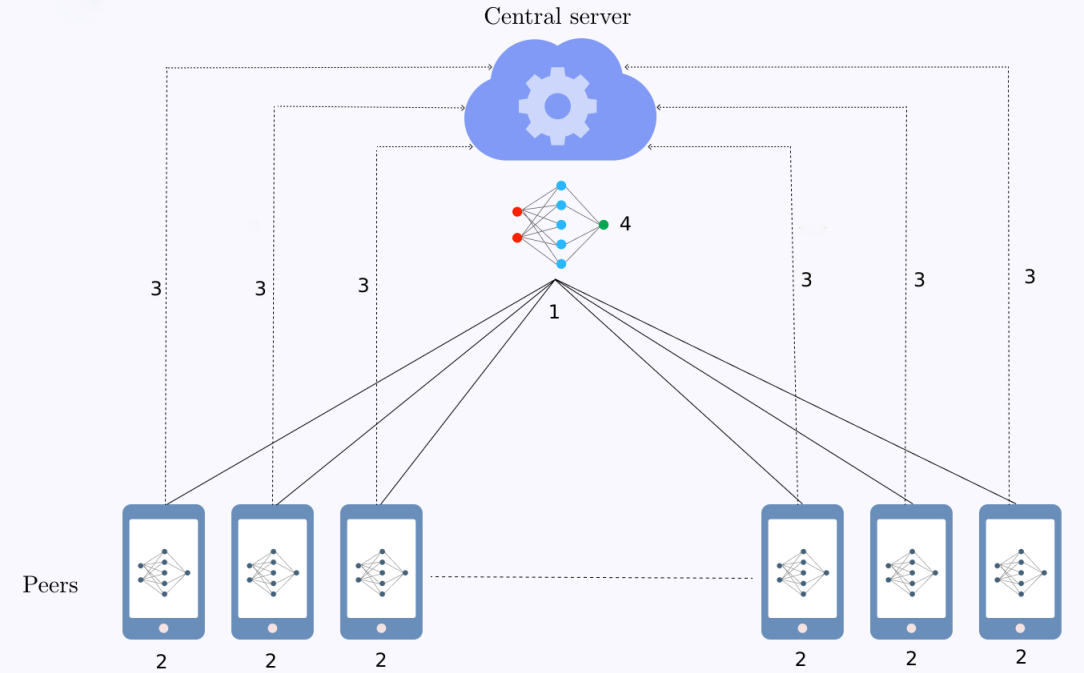

Figure 3: Federated Learning process. Source: *Devfi*

1. The server sends the global model to the clients. This step is mirrored in the base code at the beginning of the *run_experiment()* function, which is responsible for the initialization of the FL environment, and it is where the global model is initialized and sent to the clients.

2. The clients train the model with their local data. The second step exhibits its equivalence in the *participant_update()* function, which is called for each peer participating in the current round. This function is responsible for the training of the local model.

3. The clients send the updated models to the server. This step is difficult to locate in the base code, as it is not explicitly stated. However, it is possible to infer that the updated models are sent to the server in the *run_experiment()* function, where the local models are located in table structures managed by the code.

4. The server aggregates the models and sends the updated global model to the clients. This step is mirrored in the base code at the series of if statements mentioned in the previous section, 3.1, which are responsible for the aggregation of the local models.

The process is repeated until the global model converges. This step is mirrored in the loop defined in the *run_experiment()* function, which is dependant on the value of the parameter "*global_rounds*".

4 Implementation

In this section, we dive deep into our label flipping algorithms using a methodical approach that starts with a study of the base code dissected in Section 3.1. In order to lay the foundation for our hypotheses, this first step involves disassembling the existing components and exploring the rationale behind their design decisions.

On the basis of this fundamental knowledge, we present a number of novel concepts aimed towards improving and extending label flipping techniques. To push the limits of effectiveness and applicability, each modification is motivated by a strategic vision that combines theoretical insights with practical considerations.

These improvements, which range from new ideas to small modifications, collectively aim to refine and enhance the overall performance of the label flipping attacks. Through careful implementation and rigorous testing, we seek to demonstrate the efficacy of our approach and highlight the evolution of label flipping algorithms into more sophisticated and effective tools for adversarial scenarios. The implemented code can be accessed on the project's GitHub repository [15].

For the upcoming results, which will be detailed in Section 5, our focus revolves around devising attacks that optimize four parameters:

- Test error (TE). Error resulting from the loss function used in training. The lower TE, the better.
- Overall accuracy (All-Acc). Number of correct predictions divided by the total number of predictions for all the examples. The greater All-Acc, the better.
- Source class accuracy (Src-Acc). Number of the source class examples correctly predicted divided by the total number of the source class examples. The greater Src-Acc, the better.
- Attack success rate (ASR). Proportion of the source class examples incorrectly classified as the target class. Since we aim to attack the system, the higher ASR, the better.

4.1 Chosen parameters and modifications

The first step after dissecting the base code [2] is to choose the parameters that we use for our experiments:

- Independent and Identically Distributed (IID) data: In an IID case, all the peers have a uniform sample of the data, meaning each peer possesses a similar proportion of each class. Thus, each of them represents a proportion of the global data. As a result, every update they compute serves as an unbiased estimate of the global model. In the non-IID scenario, this is not the case. Each user could have different percentages of classes or contain many outliers or various other scenarios. In the most extreme case, each user might only have data corresponding to a single class. In such instances, the updates computed by peers can be significantly dissimilar from one another.

  IID is the simpler case for defenses, and therefore more challenging for attackers. If an attack performs moderately well in the IID case, it should perform better in the

non-IID case. Or, at the very least, it should be harder to detect. That is why it makes sense to start here. The non-IID case would be the most favorable scenario, as many defenses will likely fail directly. The reality is that peers typically fall somewhere in between: neither fully IID nor completely non-IID. The IID case serves as a good starting point for research into these topics.

- Global rounds: This variable is set to 100. This means that the global model is updated 100 times.
- Local epochs[3]: This variable is set to 3. This means that each peer trains its model for 3 epoch before sending the update to the server.
- Number of peers: There are 20 peers in the system.

We first began using the MNIST dataset for our experiments because it is the most common dataset used in the literature. However, we quickly realized that the results were not very interesting. The reason for this is that the MNIST dataset is too simple. The model is able to achieve a very high accuracy, and the label flipping attacks are not able to reduce it significantly. Therefore, we decided to use the CIFAR-10 dataset instead. This dataset is more complex, and the model is not able to achieve such a high accuracy. This means that the label flipping attacks are able to reduce the accuracy significantly. In the following sections, we present the results obtained with the MNIST dataset along with the ones obtained with the CIFAR-10 dataset. The reason behind this is to show the difference between both datasets. However, we focus on the results obtained with the CIFAR-10 dataset because they are more interesting. Our aim will be to flip the labels of the source class (dog) to the target class label (cat).

Another minimal change made to the base code is adding a condition checking if the device being used for the experiments is a Graphics Processing Unit (GPU) or the device's Central Processing Unit (CPU). This change is made because the GPU is much faster than the CPU when employing tensors[4]. The code used to perform this check is shown in Code 1.

Finally, we also added a section after the FL system has finished the global rounds to print the results obtained in a confusion matrix format. This matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. The code used to print this matrix is shown in Code 2.

[3]Epochs refer to the quantity of iterations a ML algorithm performs on the entire training dataset. Each epoch involves the algorithm making incremental adjustments to its model parameters based on the training data, aiming to improve its performance over time.

[4]Tensors are multi-dimensional arrays commonly used in mathematics and machine learning to represent data. In machine learning, tensors are used to store and manipulate data, such as images, sequences, and more complex structures. Tensors are particularly well-suited for ML tasks due to their ability to efficiently handle large volumes of data and perform operations in parallel. GPUs are highly effective for tensor operations, as they are optimized for parallel computing.

```
import torch
...
if torch.cuda.is_available():
        DEVICE = "cuda"
        print('GPU is available:', torch.cuda.get_device_name(0))
else:
        DEVICE = "cpu"
        print('GPU is not available, CPU will be used')
DEVICE = torch.device(DEVICE)
...
```

Code 1: Computing device selection and information

```
from matplotlib import pyplot as plt
...
actuals, predictions = self.test_label_predictions(simulation_model,
↪  self.device, self.test_loader, dataset_name=self.dataset_name)
plt.matshow(confusion_matrix(actuals, predictions))
plt.colorbar()
plt.show()
...
```

Code 2: Printing the confusion matrix

4.2 Main structure of the hypotheses code

In order to test our hypotheses, we first have to design how the code will be structured. The main idea is to have an if-statement for each hypothesis inside the *participant_update()* function, as commented in Section 3.1. This condition will contain all the code needed to perform the label flipping attack. The code will be structured as follows:

1. The first step is to iterate through the local model's samples.
   (a) For each sample, we have to check if the sample's label is the source class's label. If it is, we must keep it's index in a list.
   (b) After that, we have to make any calculation needed in order to obtain some kind of punctiuation to give to each sample. This punctuation will be used to determine which samples will be flipped.
2. Joining the index list with the punctuation list. This is helpful because we can sort the list by the punctuation and then flip the labels of the samples with the highest punctuation.
3. Determining which samples to flip based on the punctuation obtained in the previous step.

4. Flipping the labels of the samples selected in the previous step, given their indices.

5. Loading the new dataset with the flipped labels into the dataset used to train the local model.

4.3 Standard label flipping

This is the label flipping algorithm that is used in the base code [2]. It is a very simple algorithm that flips the labels of all the samples of the source class to the target class's label.

When analyzing how this algorithm works, we realize that it cannot be included in our solution as is. This is because, in a very intelligent way, it flips all the source class's labels by looping through the indices of all the samples, checking if they belong to the source class and then, flips them. As it has been stated before, our purpose is to flip a subset of these labels. Therefore, we have to devise a new algorithm that allows us to flip only the selected samples' labels. This presents us with an opportunity to craft a more targeted and sophisticated approach that aligns precisely with our objectives.

4.4 Entropy-based label flipping

The first algorithm we propose is based on the entropy of the samples. The entropy is a measure of the uncertainty of a random variable. In this case, the random variable is the label of the sample. Figure 4 shows the clear difference between a high and low entropy. The samples with a high entropy are the ones that are more difficult to classify, while the samples with a low entropy are the ones that are easier to classify.

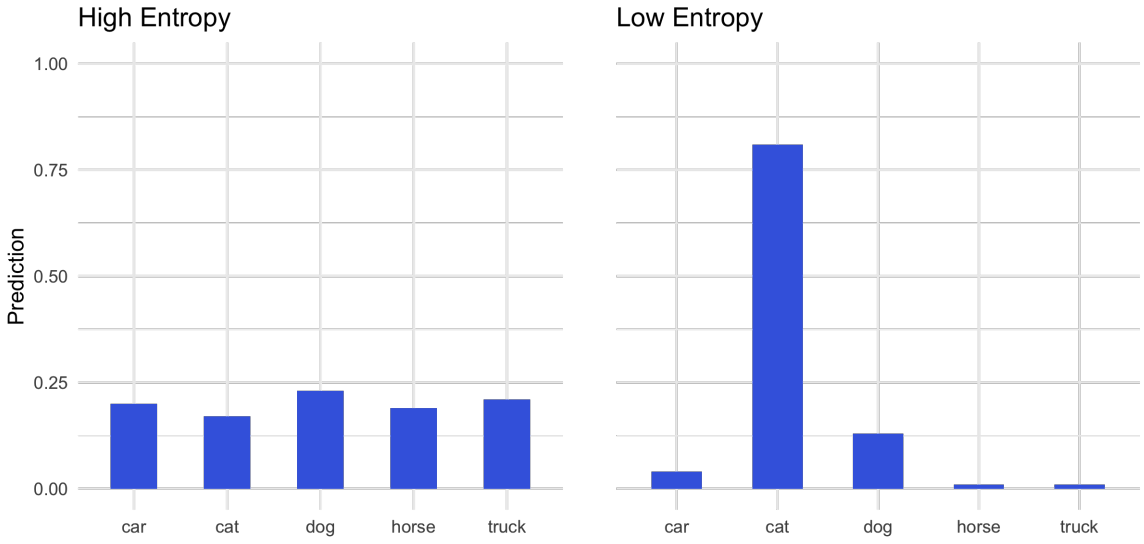

Figure 4: High vs. low entropy

Therefore, our first approach is to flip the labels of the samples with a high entropy because they are the ones that are more likely to be misclassified. The structure that this algorithm follows is the next one and the code is shown in Code 3:

1. Initialize empty lists to store the indices of the samples of the source class and the entropy associated with each sample.

2. Iterate through the local model's samples using the DataLoader[5].
   (a) For each sample, we have to check if the sample's label is the source class's label. If it is, we append it's index in the list. Since we iterate through batches of samples, it is important to keep the absolute index and not the relative one.
   (b) Then, we introduce the samples into a tensor so we can predict the output the model would return.
   (c) To obtain this output, we call the *model(data_source_batch)* function.
   (d) After that, we must transform the output to NumPy[6] format so we can calculate the entropy.
   (e) Finally, we to compute the entropy of the sample and store it in a list.
3. We merge the indices and entropies lists into a single list.
4. The next step is sorting the list by **highest entropy**.
5. We select the top 50% of the list.
6. We sort it again by index, so we can flip the labels of the samples in the correct order as we iterate in the following step.
7. To iterate through the samples in the local dataset, we create a *index_label_flip()* function, shown in Code 4 in order to overcome the problem with the main flipping function mentioned in Section 4.3. This function works as follows:
   (a) The function receives the dataset, the sorted indices and entropies list, and the target class (its ID).
   (b) It begins by the creation of an empty list to store the poisoned data.
   (c) Then, it iterates through the dataset.
   (d) For each sample, it checks if the sample's index is in the sorted indices list.
   (e) If it is, it appends a structure with the data of the sample and the target class's label to the poisoned data list.
   (f) If it is not, it appends a structure with the data of the sample and the original label to the poisoned data list.
   (g) Finally, it returns the poisoned data list.
8. Next, we create a new DataLoader object with the poisoned dataset, ready to be used to train the local model.
9. Finally, we increment a counter that keeps track of the number of attacks performed and print the information about the attack.

[5]A "DataLoader" in PyTorch refers to a utility class that simplifies the process of loading and managing data for training and testing machine learning models.

[6]NumPy is a fundamental package in the Python programming language used for numerical computations and data manipulation. It provides support for working with large, multi-dimensional arrays and matrices, along with an extensive collection of mathematical functions to operate on these arrays efficiently.

```
if (attack_type == 'entropy_label_flipping') and (self.peer_type ==
↪  'attacker'):
    train_loader = DataLoader(self.local_data, self.local_bs, shuffle =
    ↪  False, drop_last=True)
    model.eval()  # Set the model to evaluation mode
    entropies = []  # To store entropies for each data sample
    kept_indices=[] #to store the indices of the data samples with
    ↪  label==target_class
    with torch.no_grad():
        for batch_idx, (data, labels) in enumerate(train_loader):
            # Find the indices of data samples with label of the source
            ↪  class
            source_mask = labels == source_class
            # Keep the positions of the data samples with
            ↪  label==source_class
            kept_indices_batch = (batch_idx * train_loader.batch_size + i
            ↪  for i in range(len(source_mask)) if source_mask[i])
            kept_indices.extend(kept_indices_batch)
            # Get the data samples with label target
            data_source_batch = data[source_mask]
            # Send the data samples with label=source_class to the device
            data_source_batch = data_source_batch.to(self.device)
            # Obtain the predicted outputs for data samples with label
            ↪  source_class
            output = model(data_source_batch)  # Get the model's output for
            ↪  the source class data samples
            predictions_np=(output.cpu().numpy())   # Convert the output to
            ↪  numpy
            entropies.extend(stats.entropy(predictions_np, axis=1)) #
            ↪  Entropy is calculated by row
    entropy_list = list(zip(kept_indices, entropies))
    sorted_entropy_list = sorted(entropy_list, key=lambda x: x[1],
    ↪  reverse=True)    # Sort it by entropy value
    num_elements_to_keep = len(sorted_entropy_list) // 2   # Number of
    ↪  elements that represents the 50% of the data that we can attack
    top_50_percent = sorted_entropy_list[:num_elements_to_keep] # Extract
    ↪  the top 50% of the data
    sorted_entropy_list = sorted(top_50_percent, key=lambda x: x[0])  # Sort
    ↪  by index to keep the order of the data

    poisoned_data = index_label_flip(train_loader.dataset,
    ↪  sorted_entropy_list, target_class)
    # Create a new DataLoader with the updated dataset and with a shuffle
    train_loader = DataLoader(poisoned_data, self.local_bs, shuffle = True,
    ↪  drop_last=True)
    self.performed_attacks+=1
    print('Entropy-based Label flipping attack launched')
```

Code 3: Entropy-based label flipping algorithm

It is important to note that the iterations through the DataLoader are not deterministic if the *shuffle* parameter is set to true because each time we iterate through the DataLoader, the samples are shuffled. Our solution is to set the parameter to false before the first access so we can keep the indices of the samples in the same order to flip them. Then, after the dataset poisoning is completed, as we create the new DataLoader, we set the parameter to true again so the model does not learn the order of the samples.

```
1  def index_label_flip(dataset, sorted_list, target_class):
2          poisoned_data = []
3          for i, (data, label) in enumerate(dataset):
4          if i in [index for index, _ in sorted_list]:
5                  poisoned_data.append((data, target_class))  # Change the
                    ↪ label to target_class
6          else:
7                  poisoned_data.append((data, label))  # Keep the original
                    ↪ label
8          return poisoned_data
```

Code 4: *index_label_flip()* function

Given that the execution time for an attack with the parameters mentioned in Section 4.1 is approximately 4 hours for an Nvidia RTX2060 graphics card, and that we are executing 8 attacks per poisoning method, as seen in Section 2.3.5, we are not able to try all the possible modifications of the entropy-based label flipping attack. Therefore, we have to choose the ones that we think are the most promising. The modifications that we have chosen to implement and test are the following:

- Keeping the highest 25% entropies. This modification is based on the idea that the samples with the highest entropy are the ones that are more likely to be misclassified. Therefore, if we are more specific and only flip the labels of the samples with the highest entropy, we should be able to increase the attack success rate by avoiding detection from the server. To modify the code, we only have to change line 23 of Code 3 to *num_elements_to_keep = len(sorted_entropy_list) // **4***.
- Keeping the highest 75% entropies. This modification is based on the opposite idea, that is, if the attack is avoiding detection when poisoning 50% of the samples, we can try to increase the attack success rate by poisoning more samples.
- Keeping the lowest 50% entropies. The thought for this idea is that the samples with the lowest entropy are the ones that are easier to classify. Therefore, if we flip these labels, we may be able to confuse the server in the first training rounds, and indirectly lean benevolent peers to misclassify the samples according to our will. To modify the code, we only have to change line 22 of Code 3 to *sorted_entropy_list = sorted(entropy_list, key=lambda x: x[1], reverse=**False**)* so the list is ordered by ascending entropy.
- Keeping the lowest 25% entropies. The reason for this modification is the same as the

previous one, but being more specific and only flipping the labels from the samples that better represent the source class.

- Applying a scaling factor to the weights before the update. This modification is based on literature proposals to ensure a successful attack, commented in Section 2.3.5. The code used to implement this modification is shown in Code 5. The code is located at the end of the *participant_update()* function to receive the trained local model.

```
def scale_model(update, scale_factor):
        for key in update.keys():
        update[key] = (update[key].float() * scale_factor).long()
        return update
...
if self.peer_type == 'attacker' and (attack_type == 'entropy_label_flipping'
↪  or attack_type == 'closeness_label_flipping'):
        update = scale_model(model.state_dict(), scale_factor = 1.1)
        model.load_state_dict(update)
```

Code 5: Scaling factor implementation

The verification process to ensure the correct functionality of the entropy-based label flipping algorithm has been carried out with meticulous attention to detail. Extensive testing and validation procedures have been executed to ensure that the algorithm operates as intended. The different evaluations that have been used to evaluate the algorithm's behaviour and performance are listed below:

- Printing the entropies list to ensure that the range of values is correct. Then, printing the length og the entropies list and the indices list to ensure that they have the same length.
- Printing the sorted entropy list to ensure that the list is sorted by descending entropy. Then, when the list is trimmed, printing it again to make sure it is ordered by index.
- Counting and printing the amount of source class labels before and after the poisoning to ensure that the number of samples with the source class label has been reduced by the number of element in the sorted entropy list. Then, performing the same operation with the target class label to ensure that the number of samples with the target class label has been increased by the number of elements in the sorted entropy list.

This attack's code sets the base that is used for the rest of the attacks. Therefore, the testing process is the same and the code is mildly modified to adapt it to the new attack.

4.5 Closeness-based label flipping

The second proposed algorithm uses the concept of sample closeness, which serves as a metric to determine how closely a given sample aligns with classification as either the

source class or the target class. Figure 5 shows the clear difference between a high and low closeness. The samples with a high closeness are the ones that are likely to be classified as either the source or the target class, while the samples with a low closeness are the ones that are more likely to be classified as one of them.

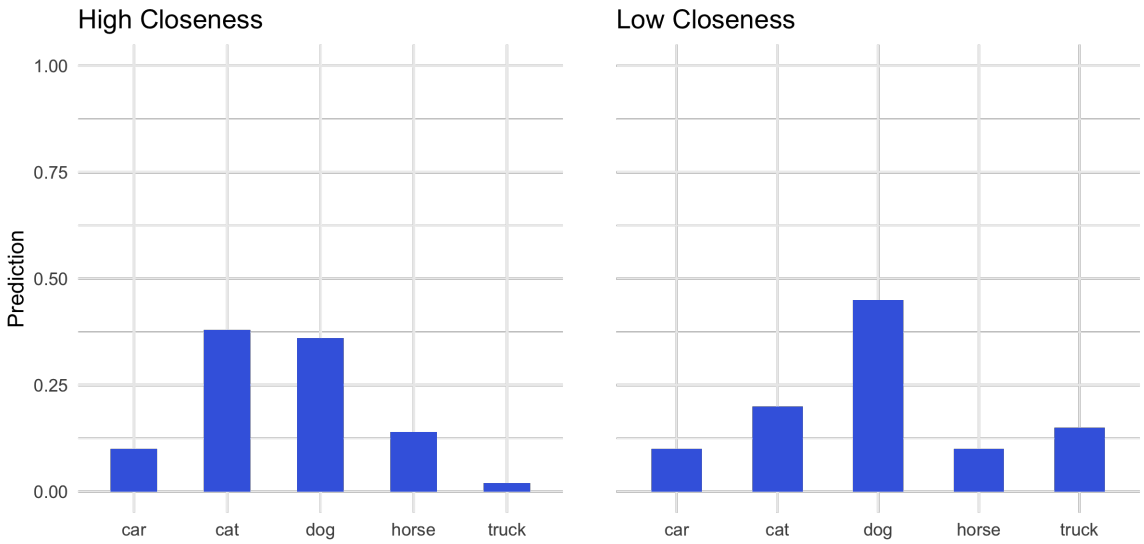

Figure 5: High vs. low closeness between classes *dog* and *cat*

Hence, our strategy involves flipping the labels of samples exhibiting high closeness, as they are more likely to suffer potential misclassification. Another reason to choose high closeness is that, the samples that have the higher closeness between the source and the target classes, may be also the ones more difficult to classify for the other peers. That could be, a dog with a short snout and hairy limbs that at first sight could be thougth of as a cat. The algorithm follows the subsequent structure, summarized since the logic is highly similar to the one explained in Section 4.4 and the corresponding code is provided in Code 6:

1. Inside the main loop, iterating through the DataLoader, instead of calculating the entropy, we calculate the closeness between the source and the target class of the sample. This is done by calculating the absolute difference between the output of the model for the source class and the target class. The output is obtained by calling the *model(data_source_batch)* function, as it was for the entropy solution.

2. The following steps remain unchanged.

```
if (attack_type == 'closeness_label_flipping') and (self.peer_type ==
↪  'attacker'):
...
closeness = []  # To store entropies for each data sample
kept_indices=[] #to store the indices of the data samples with
↪  label==target_class
with torch.no_grad():
...
        predictions_np=(output.cpu().numpy())    # Convert the output to
        ↪  numpy
        temp = []
        for i in range(len(predictions_np)):
                temp.append(abs(predictions_np[i][source_class] -
                ↪  predictions_np[i][target_class]))
        closeness.extend(temp)
...
```

Code 6: Closeness-based label flipping algorithm

The set of different modifications that we have chosen to implement and test for the closeness-based label flipping attack are the following:

- Keeping the highest 25% closeness. This modification is based on the idea that the samples with the highest closeness are the ones that are more likely to be misclassified as the target class.
- Applying a threshold instead of keeping a percentage. The rationale behind this modification is that, as the global rounds progress, the global model is more refined. Therefore, the top percentage may include samples that are not as close to the target class as they were in the first global rounds. In order to mitigate this, we can try to keep the samples with a closeness difference, lower than a threshold. The code used to implement this modification is shown in Code 7. The code is located before the call to the *index_label_flip()* function.

```
...
sorted_closeness_list = sorted(closeness_list, key=lambda x: x[1])     # Sort
↪  it by entropy value
threshold=0.02
count=1
actual=sorted_closeness_list[0][1]
while actual <= threshold and count < len(sorted_closeness_list):
        count += 1
num_elements_to_keep = count    # Number of elements that to keep
top_closeness = sorted_closeness_list[:num_elements_to_keep]
...
```

Code 7: Threshold implementation

The verifications to test the correct functionality of the closeness-based label flipping algorithm are the same as the ones used for the entropy-based label flipping algorithm.

4.6 Adaptive label flipping

The idea behind this approach is to use the previous two algorithms to create a more sophisticated and effective attack. The difference resides in the fact that, instead of using a fixed amount of samples to flip, we use a percentage of the samples depending on how far the global model has been refined, taking into account the global rounds. The algorithm follows the subsequent structure, summarized since the logic is highly similar to the one explained in Section 4.4 and the corresponding code is provided in Code 8:

1. Before the main loop, we set an if statement to check if the global rounds are greater than the first defined segment. If they are, the loop will be the same as for either the closeness-based or the entropy-based label flipping algorithm.

2. Outside the loop, we will select the amount of labels to flip depending on the global rounds. The idea is to flip a percentage of the samples that is inversely proportional to the global rounds. This means that, as the global rounds increase, the percentage of samples to flip decreases.

3. If the current global round resides on the first segment, the code will be the same as for the standard label flipping algorithm. This means that all the source class labels will be flipped to the target class label.

```
if (attack_type == 'stealthy_closeness_label_flipping') and (self.peer_type == 'attacker'):
        ...
        if global_epoch > 25:
                with torch.no_grad():
                        ...
                sorted_entropy_list = sorted(entropy_list, key=lambda x: x[1], reverse=True)
                num_elements_to_keep=0
                if global_epoch <= 50:  # round 25 to 50, flip 75% of the data
                        num_elements_to_keep = len(sorted_entropy_list) // 2 + len(sorted_entropy_list) // 4
                if global_epoch <= 75:  # round 50 to 75, flip 50% of the data
                        num_elements_to_keep = len(sorted_entropy_list) // 2
                if global_epoch <= 100: # round 75 to 100, flip 25% of the data
                        num_elements_to_keep = len(sorted_entropy_list) //4
                ...
        else:   # round 0 to 25, flip all the data
                poisoned_data = label_filp(self.local_data, source_class, target_class)
                train_loader = DataLoader(poisoned_data, self.local_bs, shuffle = True, drop_last=True)
        ...
```

Code 8: Adaptive label flipping algorithm

The first segment does not follow the usual code because, since all the labels are flipped there is no point in computing the entropy or the closeness of the samples.

The set of different modifications that we have chosen to implement and test for the adaptive label flipping attack are the following:

- Testing with entropy and closeness-based label flipping to flip all the labels during the first half of the global rounds. During the second half, flip 50% of the samples.
- Testing with either entropy or closeness-based label flipping to flip all the labels during the first half of the global rounds. During the second half, flip 25% of the samples.

The verifications to test the correct functionality of the adaptive label flipping algorithm are the same as the ones used for the previous label flipping algorithms.

5 Results

intro to the section, what are we going to talk about? com s'ha explicat a ... l'objectiu és maximizar all acc i asr

explicar main problem és el temps d'execució

5.1 Results for entropy-based label flipping

What is it? where can it be found? pros?

6 Conclusions

intro to the section, what are we going to talk about? Més enllà (i això pot anar a la secció de conclusions i treball futur), es miraria com afecten els atacs en el cas non-iid.

6.1 Project's architecture

architecture of Najeeb's code, where are my functions? scheme on how the system (server, epochs, peers, peer rounds) works

6.2 Future work

non-iid seria mes chetao per un atac com s'ha comentat a impl/modificacions(4.1)

References

[1] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction, 2019.

[2] Najeeb Jabreel. Lfighter, 2023. URL https://github.com/NajeebJebreel/LFighter.

[3] Md. Omaer Faruq Goni, Fahim Md. Sifnatul Hasnain, Md. Abu Ismail Siddique, Oishi Jyoti, and Md. Habibur Rahaman. Breast cancer detection using deep neural network, 2020.

[4] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2017.

[5] European Commission. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance), 2016. URL https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[6] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates, 2021.

[7] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent, 2017.

[8] Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. The limitations of federated learning in sybil settings, 2020.

[9] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. Data poisoning attacks against federated learning systems, 2020.

[10] Thien Duc Nguyen, Phillip Rieger, Roberta De Viti, Huili Chen, Björn B Brandenburg, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, et al. {FLAME}: Taming backdoors in federated learning, 2022.

[11] Najeeb Moharram Salim Jebreel et al. Protecting models and data in federated and centralized learning.

[12] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[13] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). URL http://www.cs.toronto.edu/~kriz/cifar.html.

[14] Aditya Pal, Abhilash Barigidad, and Abhijit Mustafi. Imdb movie reviews dataset, 2020. URL https://dx.doi.org/10.21227/zm1y-b270.

[15] Eduard Bel. Master's thesis code, 2023. URL https://github.com/EduardBel/MastersThesisCode.

7 Example title

7.1 Example subtitle

7.1.1 Example subsubtitle

8 Text examples

Example text example text

8.1 Bold & italic text

Bold text

Italic text

8.2 In document refernces

Equation 1

8.3 Other documents refernce

referenced text[?] test cite [?]

8.4 Acronyms & footnotes

ACRONYM[7]

Footnote[8]

8.5 Hyperlinks / URLs

Exemple of named hyperlink

[7] Acronym text
[8] Footnote text

9 Example lists

9.1 Unordered list

- Item
- Item
- Item

9.2 Ordered list

1. Item
2. Item
3. Item

10 Equation example

$$a^b = c \tag{1}$$

11 Table example

	API Disponible	API Obsoleta	Dificultat	Característiques avançades
Camera	1	21	Senzilla	No
CameraX	21	N/A	Senzilla	Sí[9]
Camera2	21	N/A	Complexa	Sí

Table 1: Comparativa d'APIs de càmera

As we can see in Figure 6, it works

12 Image example

Figure 6: Logo URV

13 Code snippet example

For all minted listings is required to enable *-shell-escape* on the LaTeX executable and have pygments installed

```
import numpy as np

def incmatrix(genl1,genl2):
    m = len(genl1)
    n = len(genl2)
    M = None #to become the incidence matrix
    VT = np.zeros((n*m,1), int)  #dummy variable

    #compute the bitwise xor matrix
    M1 = bitxormatrix(genl1)
    M2 = np.triu(bitxormatrix(genl2),1)
...
```

Code 9: Code example

14 Diagram examples

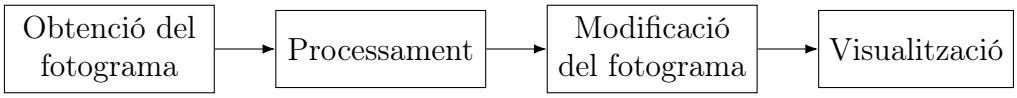

Figure 7: Projecte workflow

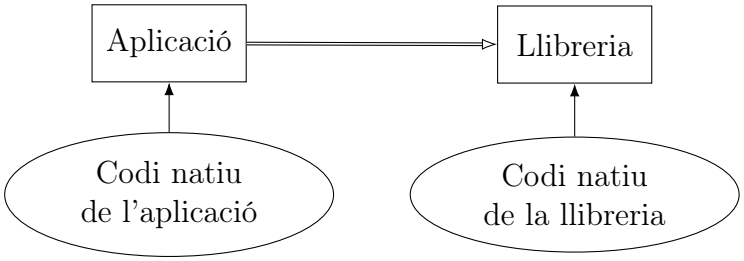

Figure 8: Module dependency

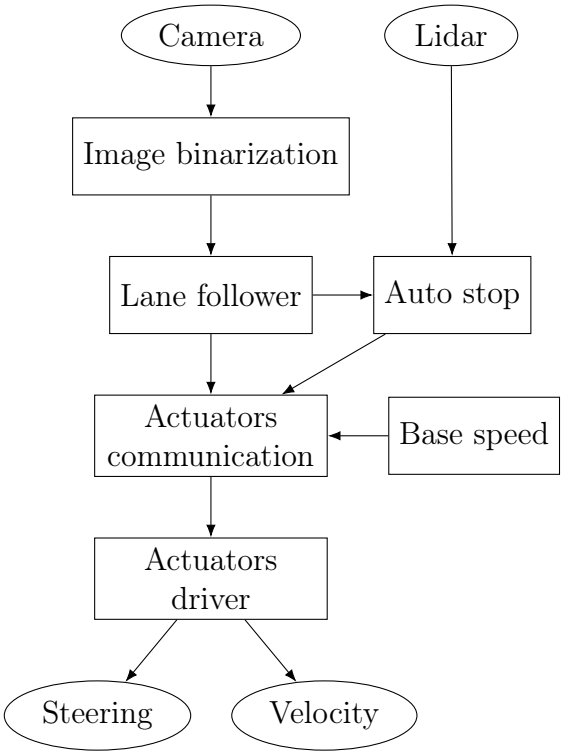

Figure 9: Car nodes layout

Appendix A:
Apendix example