



Degree project in Mathematical Statistics

Second cycle, 30 credits

Label leakage from Regression Models Gradients in Federated Learning

[Click here to enter your subtitle](#)

JEAN LEPROVOST

Abstract

Federated learning (FL) is one of the most popular way to collaboratively train models while preserving data privacy. Participants train their model locally and share only their gradients instead of their personal data. However, recent gradient attacks have shaken this guarantee of "privacy by design" by reconstructing the participants data from the shared gradients. Serious improvements have been achieved by first inferring the labels of the data, making it easier to then reconstruct the input data. Until now these attacks have been studied only in the context of classification models, leaving the regression case unaddressed. In this paper we develop a gradient-based attack on labels in the context of a regression model being trained under a FL framework. This attack relies on solving an approximated linear system of equations of gradients and labels, calibrated using auxiliary data. Our experiments show promising results about inferring labels considering a FL regression model.

Abstract

Federated learning (FL) är ett av de mest populära sätten att gemensamt träna modeller med bibehållen integritet. Deltagarna tränar sin modell lokalt och delar bara sina gradienter istället för sina personuppgifter. Nya gradientattacker har dock skakat denna garanti för "privacy by design" genom att rekonstruera deltagarnas data från de delade gradienterna. Stora förbättringar har uppnåtts genom att först härleda datans etiketter, vilket gör det lättare att sedan rekonstruera indata. Hittills har dessa attacker endast studerats i samband med klassificeringsmodeller, vilket innebär att regressionsfallet inte har behandlats. I det här dokumentet utvecklar vi en gradientbaserad attack på etiketter i samband med en regressionsmodell som tränas under ett FL-ramverk. Denna attack bygger på att lösa ett approximerat linjärt system av ekvationer av gradienter och etiketter, kalibrerade med hjälp av hjälpdata. Våra experiment visar lovande resultat när det gäller att härleda etiketter med hänsyn till en FL-regressionsmodell.

Contents

1	Introduction	3
1.1	Federated Learning	3
1.1.1	Definition	3
1.1.2	FL challenges	3
1.2	Privacy in Federated Learning	4
1.2.1	Inference Attacks	5
1.2.2	Privacy Techniques	5
1.3	Aim	6
2	Background	7
2.1	Neural networks	7
2.1.1	The Multilayer Perceptron	7
2.1.2	Gradients and SGD	7
2.1.3	Classification vs. Regression model	8
2.2	Federated learning	9
2.2.1	FedSGD & FedAvg	9
3	Related work	11
3.1	Gradient attacks	11
3.1.1	DLG	11
3.2	Label inference	12
3.2.1	iDLG	13
3.2.2	BLR (Batch Label Restoration)	14
3.2.3	LLG	15
3.2.4	iLRG	16
3.3	Regression	17
3.3.1	Motivation	17
3.3.2	Our problem setup	18
4	Methods	19
4.1	The $K = 1$ case	19
4.1.1	An Exact Solution	19
4.1.2	Generalization to other activation functions	20
4.2	General case : $K > 1$	20
4.2.1	A basic attack: mean y estimation	20
4.2.2	Problem transformation	21
4.2.2.1	metaclasses	21
4.2.2.2	The \bar{y} attack	21
4.2.3	The k attack	22

4.2.4	Approximations	23
4.3	Linear Least squares problem	23
4.3.1	System of linear equations	23
4.3.2	Least squares estimate	24
4.3.3	The Moore-Penrose inverse	24
4.3.4	Regularization techniques	25
4.3.4.1	Tikhonov regularization	25
4.3.4.2	Choice of beta : the L-curve	26
5	Data	28
5.1	Synthetic dataset	28
5.2	Salary dataset	28
6	Results	30
6.1	Protocol	30
6.2	$K=1$ reconstruction	30
6.3	Basic attack	31
6.4	\bar{y} attack	32
6.4.1	Synthetic dataset	33
6.4.2	Salary dataset	33
6.5	k attack	34
6.6	Robustness experiments	35
6.6.1	Batch size K	35
6.6.2	Last layer size M	35
6.6.3	Auxiliary data fit	36
7	Conclusion	40
A		42

Chapter 1

Introduction

In today’s highly interconnected world, the vast amount of data and user information has integrated Machine Learning (ML) techniques into everyday life and numerous services. Most ML methods operate in a centralized manner, often necessitating the collection and processing of large volumes of user data by central service providers. This data can be sensitive, leading to concerns about whether it is managed in line with user expectations and privacy regulations, such as the European General Data Protection Regulation (GDPR).

1.1 Federated Learning

1.1.1 Definition

Federated learning (FL) [4] is a ML approach allowing to train a model on various user data without accessing their personal data. Concretely, a central coordinator called the *server* first sends an untrained model to the participants who hold the data, called the *clients* (typically hospitals or personal devices like smartphones, holding personal data that can be sensitive). Then each client train the model locally on their personal data and send back the updated model (or possibly the gradients) to the server. These updated models are then aggregated together by the server, *e.g.* by computing the mean or median of the parameters. This pattern called a round of communication is repeated successively as many times as needed to train the model (Figure 1.1). Therefore, privacy seems guaranteed “by design” since clients never share their personal data directly but only their model updates.

We can consider two main use cases for the FL framework, called cross-device and cross-silo. In the cross device case, the problem is massively distributed, resulting in a number of clients much larger than the average number of data samples per client. This is specifically the case when considering a model train on many personal smartphones, each holding personal data. On the opposite, the cross-silo case considers a few clients each holding many samples, *e.g.* when different hospitals (the clients) collaboratively train a model on their patients’ data (the samples).

1.1.2 FL challenges

Training a model under a distributed learning frameworks like FL raises various specific challenges compared to the centralized setting.

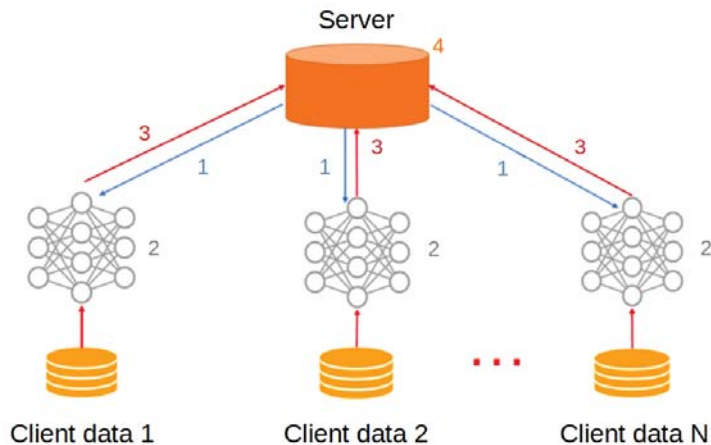


Figure 1.1: The federated learning framework. *The model is sent by the server to the clients (1) who train it locally on their own data (2), then updated models are sent back to the server (3) who aggregate them (4).*

Non-IID unbalanced data. In the FL cross-device especially, the data is typically assumed to be non-independent and identically distributed (non-IID) among clients, since clients have different personal behavior. Data can also be unbalanced, *i.e.* clients don't have the same amount of samples. Therefore, it is not obvious that the optimization problem of learning a model under FL converges as well as in the centralized manner.

Communication. FL has the indubitable advantage of distributing the training computations (gradient computations) over the clients, thus reducing drastically computation costs for the server. The main cost of FL are therefore the communication costs between the server and the clients. One way to mitigate this cost is to increase computation while decreasing communication, *e.g.* by adding local computation steps for each client. Moreover, additional communication problems can be faced in the cross-device case, since some clients can be non-available anytime.

Model Poisoning. The FL design introduces several attack surfaces that adversaries can exploit. In model poisoning attacks, malicious clients (also called Byzantine attackers) deliberately send manipulated updates to corrupt the global model. This can be done to degrade the overall performance of the model or to introduce specific biases. In the Sybil attack scenario, for example, a malicious user simulates numerous fake client devices to alter the training.

1.2 Privacy in Federated Learning

Since the clients never share their personal raw data, Federated Learning (FL) appears as a solution guaranteeing privacy by its design. However, recent work has shaken this belief, showing how client data can be reconstructed from the shared updates or gradients by an honest-but-curious server.

1.2.1 Inference Attacks

These attacks aim at inferring information on clients' dataset by analyzing the model updates at different iterations. Since the communication of the updates can easily be encrypted between the clients and the server, such attacks are more likely performed by an *honest-but-curious* server who follows the prescribed FL protocol but attempts to reconstruct private client data from the received updates or gradients. The three main types of inference attacks are:

- **Membership Inference Attack:** the server identify whether a data sample was in the client dataset.
- **Property Inference Attack:** Infers other properties of the data that are unrelated to the primary learning objective.
- **Reconstruction Attack:** Attempts to reconstruct the client dataset (either inputs or labels, or both).

These attacks are particularly concerning as it questions the inherent privacy guarantees that FL is supposed to offer. Recent techniques have demonstrated the feasibility of such attacks, making them a critical area of concern.

1.2.2 Privacy Techniques

To mitigate these privacy risks, several defense mechanisms have been proposed. These techniques can broadly be categorized into cryptographic methods and defenses based on information theory.

Cryptographic Methods

- **Homomorphic Encryption** allows computations to be performed on encrypted data without decrypting it. This ensures that the server can aggregate or update the global model without ever accessing the raw data or sensitive gradients.
- **Secure Multi-Party Computation (SMPC)** enables multiple parties to jointly compute a function over their inputs while keeping those inputs private. In the context of FL, SMPC can be used to securely aggregate model updates from multiple clients without exposing their individual data to the server or each other.

While these cryptographic methods offer strong security guarantees, they can be computationally expensive and may introduce significant communication overhead, making them challenging to implement in large-scale FL systems.

Information-Theoretic Defenses

- **Differential Privacy (DP)** works by adding carefully calibrated noise to the updates or gradients before they are shared with the server. This noise ensures that the contribution of any single client's data to the global model is obfuscated, thus protecting the privacy of individual

data points. DP provides a formal privacy guarantee, which ensures that the output of the FL process does not significantly change when a single client's data is included or excluded.

Information-theoretic defenses like DP are often favored for their strong theoretical guarantees and flexibility. However, there is often a trade-off between privacy and model performance, as too much noise can degrade the accuracy of the model.

1.3 Aim

The goal of this thesis is to study the feasibility of a reconstruction attack on labels in a FL framework where a regression model is trained. After presenting the basic algorithms of machine learning and federated learning, we review the label inference attacks that have been developed in the case of classification models. We then explore the unaddressed case of regression models, developing our own attacks to leak the labels or other data properties. Finally, we evaluate these attacks on various datasets and assess their effectiveness.

Chapter 2

Background

2.1 Neural networks

Neural networks (NNs) are a widely used model architecture in ML for their ability to recognize a wide variety of patterns in data. A neural network is composed of many interconnected neurons. Basically, each neuron receives the outputs of preceding neurons, apply basic linear and non-linear operations to it, and transmit this output to following neurons. There exists a large variety of neurons and model architectures (MLP, CNNs, RNNs, transformers) depending on the type of data (tabular, image, audio, text) and the task of the model (classification, regression, generation). For example, convolutional neural networks (CNNs) are more adapted to image data.

2.1.1 The Multilayer Perceptron

One of the simplest NNs is arguably the Multilayer Perceptron (MLP). This model consist of several “hidden” layers of fully connected (FC layers) neurons with non linear activation functions (Figure 2.1).

Concretely, each neuron computes a weighted averaged sum of the outputs of the previous layer neurons (according to the weight parameters w), adds a bias b to this sum and apply a non linear activation function σ such as the sigmoid or ReLU function.

$$output = \sigma(w^T inputs + b) \tag{2.1}$$

Although this model is simple and not representative of the wide variety of model architecture, most NNs and even complex ones often end with one or a few FC layers like the MLP.

2.1.2 Gradients and SGD

The training of a NN consist in optimizing its parameters (weights and biases) such that the model outputs the true label given an input datum. To be more precise, we define a loss function $\mathcal{L} : (z, y) \mapsto \mathcal{L}(z, y)$ that computes the loss (the “error”, that is to be minimized) between the output prediction of the model z and the ground-truth label y . In supervised learning, this is done by iteratively computing the loss of the model on labelled data samples, and optimizing the parameters to minimize this loss. Concretely, for each

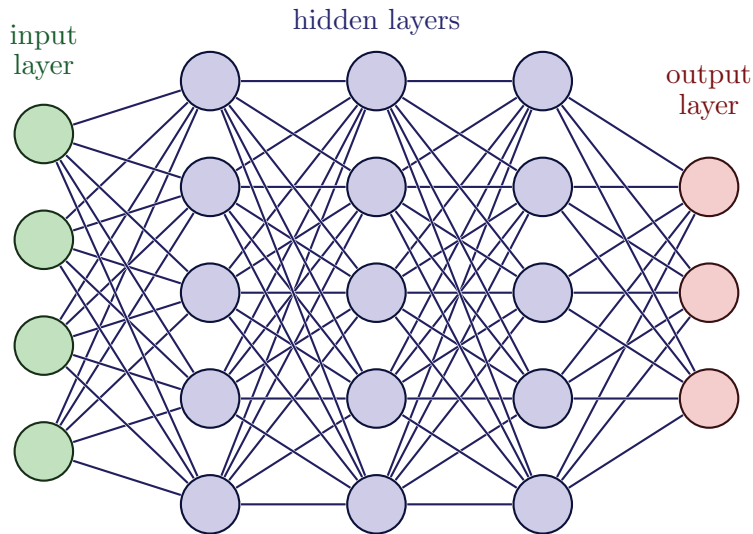


Figure 2.1: A simple MLP

sample-label pair (x, y) we compute the gradient ∇p of the loss according to the parameters, and update the parameters by stepping in the opposite direction of the gradient.

$$p_{t+1} = p_t - \eta \nabla p_t, \quad \text{with } \nabla p_t := \frac{\partial \mathcal{L}(\text{model}(x), y)}{\partial p_t} \quad (2.2)$$

The hyper parameter η is called the learning rate, and control the speed of converging. Thanks to the structure of NNs, gradients are computed efficiently by leveraging the chain rule to back propagate the gradient through the layers.

SGD. Most neural network models are trained using variants of the stochastic gradient descent (SGD) to optimize their parameters. SGD consists in computing the batch-averaged gradient of the loss according to the parameters over a randomly chosen mini-batch (or simply “batch”) of K sample-label pairs among the whole dataset. Namely, it consists in taking $\nabla p_t := \frac{1}{K} \sum_{k=1}^K \frac{\partial \mathcal{L}(\text{model}(x^k), y^k)}{\partial p_t}$ in equation 2.2. Using mini-batch gradient descent provides a good trade-off between precision and speed for an efficient optimization process.

2.1.3 Classification vs. Regression model

Classification and regression models differ primarily in the type of output they produce, the nature of the last layer in a neural network, the type of labels they handle, and the loss functions they use. In a **classification model**, the task is to predict discrete class labels. The labels y are represented as one-hot encoded vectors for N classes, meaning that if a given sample (x, y) belongs to class $c \in \{1, \dots, N\}$, then $y \in \mathbb{R}^N$ and

$$y_i = \begin{cases} 1 & \text{if } i = c \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

The output of the last layer is a vector of length N , and an activation function such as the softmax function is applied to this vector to produce a probability

distribution over the classes:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^N e^{z_j}} \quad (2.4)$$

where z_i is the i -th element of the output vector. The predicted class is the one with the highest probability. A common loss function used in classification tasks is the **cross-entropy loss**, defined as:

$$\mathcal{L}_{\text{CE}}(z, y) = - \sum_{i=1}^N y_i \log(\text{softmax}(z_i)) = - \log \frac{e^{z_c}}{\sum_{j=1}^N e^{z_j}} \quad (2.5)$$

since $y_i = 1$ if $i = c$, and 0 otherwise.

On the other hand, a **regression model** aims to predict continuous values. The labels y in this case are real-valued numbers. The output of the last layer in a regression neural network is typically a single neuron (or multiple neurons for multivariate regression) without an activation function or with a linear activation function:

$$\hat{y} = z \quad (2.6)$$

where z is the direct output of the final layer. A commonly used loss function for regression tasks is the **Mean Squared Error (MSE)**, defined as:

$$\mathcal{L}_{\text{MSE}}(z, y) = \frac{1}{K} \sum_{k=1}^K (z_k - y_k)^2, \quad (2.7)$$

where z_k is the predicted value and y_k is the true label of the sample k in a batch of K samples.

2.2 Federated learning

The exact training process of a model in FL can vary depending on the context. For example, we can choose to take a more or less important fraction of the clients participating at each round, which is necessary if all the clients are not available. We can also set how each client updates the model locally on its own dataset. More precisely, we can choose the batch size K , the number of epochs E , and the learning rate η to define the client's local training process.

2.2.1 FedSGD & FedAvg

FedSGD is arguably the most basic FL algorithm. In this case, each client performs a single SGD computation over its whole dataset ($K = D$ and $E = 1$, where D is the dataset size of the client), and then transmit their updated model to the server. Concretely, for a number C of clients, and p_t the model parameters at communication round t :

$$p_{t+1} = \frac{1}{C} \sum_{n=1}^C p_{t+1}^{(n)} \quad (2.8)$$

Since the updated model of client n is simply $p_{t+1}^{(n)} = p_t - \eta \nabla p_t^{(n)}$, we can rewrite the equation in:

$$p_{t+1} = p_t - \frac{\eta}{C} \sum_{n=1}^C \nabla p_t^{(n)} \quad (2.9)$$

Therefore, we note that it is equivalent that clients directly share their gradients $\nabla p_t^{(n)}$ rather than the model updates $p_{t+1}^{(n)}$. Reciprocally, since the server know the initial model p_t and the learning rate η , it can reconstruct client n parameters' gradient $\nabla p_t^{(n)} = \frac{p_{t+1}^{(n)} - p_t}{\eta}$.

A more advanced algorithm used in federated learning is **FedAvg**, which was introduced in 2016 in [4]. In this protocol, clients perform several optimization steps locally before returning their model update. Similarly to a centralized data training, each client splits its dataset in batches of size K , and train the model over multiple epochs ($E > 1$). This algorithm increases the number of computations ($D/K \times E$ gradients computed vs. 1 for FedSGD) but has the advantage of reducing communication costs since fewer communication rounds will be needed for the model to converge.

Chapter 3

Related work

3.1 Gradient attacks

Although federated learning seems to ensure the clients data privacy by its design, it has been shown that this framework is not immune. For example, since clients share their updated models to the server, this one could compare this updated model to the initial model and try to reconstruct the training data that led to this update. This is exactly how gradient attacks work, a honest but curious server is faced with the following inverse problem : finding the training data from the shared gradients.

Note : When using FedSGD, we showed in 2.2.1 that clients can alternatively share their model updates or their gradients. Therefore, the term gradient attacks initially referred to the case where FedSGD was used, but was then also employed to attacks on updates (*e.g.* with FedAvg, updates are shared and not gradients).

3.1.1 DLG

The first gradient attack was exposed in 2019 in the article *Deep Leakage from Gradient* (DLG) [8] by Zhu et al. They propose a method to reconstruct both the inputs x and the labels y of a client dataset from its shared gradient. Briefly, the attack consists in an optimization process that finds the best input-label pairs producing a gradient that best matches the shared gradient.

Consider the model $F()$ that is trained in a FL context with FedSGD, using parameters p and taking an input data x to output a prediction $F(x, p)$. At each training round t , the server sends parameters p_t to the clients. Then client n computes and sends back the following gradient:

$$\nabla p_t^{(n)} = \frac{\partial \mathcal{L}(F(x_t^{(n)}, p_t), y_t^{(n)})}{\partial p_t} \quad (3.1)$$

Since the server knows by default the initially sent parameters p_t and model $F()$, it can construct a “dummy gradient” $\nabla p' = \frac{\partial \mathcal{L}(F(x', p_t), y')}{\partial p_t}$ from randomly initialized dummy data (x', y') . If this dummy gradient $\nabla p'$ is close to the true gradient $\nabla p_t^{(n)}$, then the dummy data (x', y') will be close to the true data $(x_t^{(n)}, y_t^{(n)})$. Therefore, DLG consists in finding a dummy gradient that

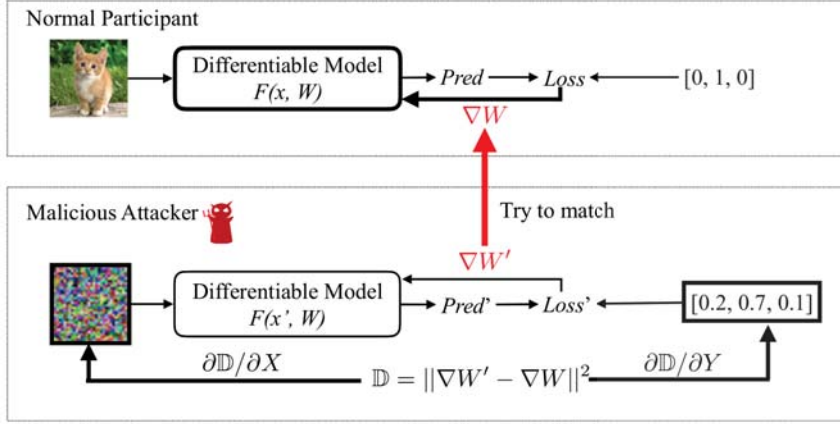


Figure 3.1: The DLG method (extracted from [8]). The malicious attacker adjusts its dummy inputs and labels (in bold) to match its dummy gradient $\nabla W'$ with the true gradient ∇W .



Figure 3.2: DLG reconstruction of a batch of 8 samples (extracted from [8]). The dummy inputs are randomly initialized, and are fully leaked after 2711 iterations (up to a permutation). The model used was a ResNet-20.

matches the real leaked gradient by solving the following optimization problem:

$$x'^*, y'^* = \operatorname{argmin}_{x', y'} \|\nabla p' - \nabla p_t^{(n)}\| \quad (3.2)$$

This problem is solved iteratively by gradient descent (see Figure 3.1). First, a dummy input data x' and dummy label y' are randomly initialized according to a normal distribution. Since the server knows by default the initially sent parameters p_t and the model $F()$, a dummy gradient can be computed from this dummy data $\nabla p' = \frac{\partial \mathcal{L}(F(x', p_t), y')}{\partial p_t}$. The distance \mathbb{D} between this dummy gradient and the true shared gradient is then computed. Finally, the dummy data and labels are updated to minimize this distance, namely $x' \leftarrow x' - \eta \nabla_{x'} \mathbb{D}$ and $y' \leftarrow y' - \eta \frac{\partial \mathbb{D}}{\partial y'}$.

This method gives impressive reconstruction results, as we can see on Figure 3.2 where a full image batch is reconstructed with pixel-wise accuracy.

3.2 Label inference

One way to improve a gradient-based attack like DLG is to first reconstruct the labels y . In this way, there are only the inputs x left to find, which

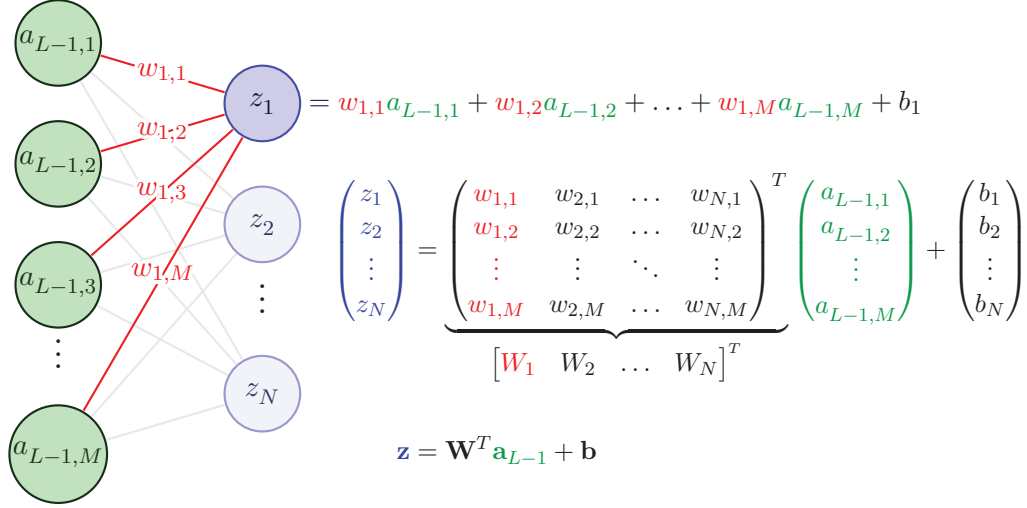


Figure 3.3: Notations for the last fully connected layer.

simplifies the problem. All the following attacks are based on an analysis of the parameters' gradients of the last layer, that is always assumed to be a fully connected layer.

Note: the notations for this section are grouped on Figure 3.3.

3.2.1 iDLG

In 2021, [7] proposed an analytical method to reconstruct the label from the gradient. Their method, iDLG (Improved Deep Leakage from Gradients), is an analytical approach that reconstructs exactly the label y from a gradient computed on a batch of size $K = 1$ (containing a unique sample).

They consider a classification model trained with FedSGD, where a client has a dataset containing a single labelled sample (x, y) . In classification (*cf.* §2.1.3), the model prediction z is compared to the label y using the cross entropy loss function:

$$\mathcal{L}(z, y) = -\log \frac{e^{z_c}}{\sum_i e^{z_i}} \quad (3.3)$$

Where c is the true class of the sample among the N classes and $z \in \mathbb{R}^N$. Considering a model of L layers, the trick is to observe the gradient of the weights W_i connecting layer $L - 1$ to the i -th output of the output layer (last layer L). This can be done using the chain rule:

$$\nabla W_i = \frac{\partial \mathcal{L}(z, y)}{\partial z_i} \frac{\partial z_i}{\partial W_i} \quad (3.4)$$

Since $z_i = W_i^T a_{L-1} + b_i$, the second factor is simply a_{L-1} the output ("activation") of the before to last layer $L - 1$. The first factor is more interesting,

we name it d_i :

$$d_i = \frac{\partial \mathcal{L}(z, y)}{\partial z_i} = -\frac{\partial \log e^{z_c} - \partial \log \sum_j e^{z_j}}{\partial z_i} \quad (3.5)$$

$$= -\frac{\partial z_c}{\partial z_i} + \frac{\frac{\partial}{\partial z_i} \sum_j e^{z_j}}{\sum_j e^{z_j}} \quad (3.6)$$

$$d_i = \begin{cases} -1 + \frac{e^{z_c}}{\sum_j e^{z_j}} & \text{if } i = c \\ \frac{e^{z_i}}{\sum_j e^{z_j}} & \text{otherwise} \end{cases} \quad (3.7)$$

Since $\frac{e^{z_i}}{\sum_j e^{z_j}} \in (0, 1) \quad \forall i \in \{1, \dots, N\}$, then we have $d_i \in (-1, 0)$ when $i = c$ and $d_i \in (0, 1)$ otherwise. Now since $\nabla W_i = d_i \cdot a_{L-1}$, we can find c using the following criterion:

$$i = c \Leftrightarrow \nabla W_i \cdot \nabla W_j \leq 0, \quad \forall j \neq i \quad (3.8)$$

If we suppose that a_{L-1} is non-negative which is the case if a non-negative activation function is used *e.g.* ReLU or Sigmoid, then we just need to look at the sign of each ∇W_i for $i \in \{1, \dots, N\}$ and find the one with negative elements.

Extracting the true label with this method before performing DLG increases the fidelity of reconstructed data. It also makes it easier for DLG to converge (*e.g.* 90 iterations instead of 200 [7]).

3.2.2 BLR (Batch Label Restoration)

The main limitation of the iDLG method is that it only allows reconstructing a batch of size $K = 1$. In 2021, Yin et al. presented an approach in [6] that extends this method to a batch of size $K > 1$ where there is at most one sample per class.

Consider a batch of K input-label pairs $(\mathbf{x}, \mathbf{y}) = \{(x_k, y_k)\}_{k=1}^K$. Taking the same notations as in §3.2.1, the gradient of the weights connecting layer $L - 1$ to the i -th output is now averaged over the K samples:

$$\nabla W_i = \frac{1}{K} \sum_k \nabla W_i^k = \frac{1}{K} \sum_k d_i^k a_{L-1}^k \quad (3.9)$$

Taking the sum of this gradient vector, we construct the indicator s_i :

$$s_i := \mathbf{1}^T W_i = \frac{1}{K} \sum_k d_i^k \mathbf{1}^T a_{L-1}^k \quad (3.10)$$

As shown in §3.2.1, $d_i^k \in (-1, 0)$ if $c_k = i$ and $d_i^k \in (0, 1)$ otherwise, noting c_k the true class label of sample k . Therefore, assuming a non-negative activation function, if $s_i < 0$ then there is a sample of class i in the batch (only one as assumed before). On the other hand, if there is one sample k_i belonging to class i , then s_i is not necessarily negative since the negative sign of $d_i^{k_i}$ could be lost in the summation. However, the authors observe empirically that $|d_i^{k_i} \mathbf{1}^T a_{L-1}^{k_i}| \gg |d_i^{k \neq k_i} \mathbf{1}^T a_{L-1}^{k \neq k_i}|$, which leaves the negative sign mostly intact

when bringing positive values from the other samples. Therefore, one can use the following method to extract the K class index labels \hat{c} :

$$\hat{c} = \underset{m}{\operatorname{argsort}}(\min \nabla W_i)[: K] \quad (3.11)$$

Where $\nabla W = [\nabla W_1, \dots, \nabla W_N] \in \mathbb{R}^{M \times N}$ is the shared gradient of the last layer weights. The minimum element of the vector ∇W_i is taken (instead of the sum $\mathbf{1}^T$) to be more robust to the ‘‘pollution’’ of positive signs (see Figure A.1 for a more visual explanation of this method).

3.2.3 LLG

Once again, there is a limitation to the previous method since it assumes that all labels are different in the batch. The LLG method (*Label Leakage from Gradients*) developed in 2022 in [5] overcomes this limitation. The method is based on the following decomposition of d_i :

$$d_i = \frac{\partial \mathcal{L}(Z, Y)}{\partial z_i} = \frac{1}{K} \sum_k \frac{\partial \mathcal{L}(z^k, y^k)}{\partial z_i^k} = \frac{1}{K} \sum_k d_i^k \quad (3.12)$$

$$= \frac{1}{K} \sum_k \left(-\delta_{c_k, i} + \frac{e^{z_i^k}}{\sum_j e^{z_j^k}} \right) \quad \text{with } \delta_{c_k, i} = \begin{cases} 1 & \text{if } c_k = i \\ 0 & \text{otherwise} \end{cases} \quad (3.13)$$

$$d_i = -\frac{\lambda_i}{K} + \frac{1}{K} \sum_k \frac{e^{z_i^k}}{\sum_j e^{z_j^k}} \quad (3.14)$$

Where λ_i is the number of samples of class i in the batch. Since the second term is in $(0, 1)$, (3.14) shows that d_i is smaller the higher λ_i is. Once again, we don’t have access to d_i directly in the shared gradients but to $\nabla W_i = d_i \cdot \bar{a}_{L-1}$ for all i . Let’s define the indicator $g_i = \mathbf{1}^T \nabla W_i$:

$$g_i = d_i (\mathbf{1}^T \bar{a}_{L-1}) \quad (3.15)$$

$$= \underbrace{-\frac{\mathbf{1}^T \bar{a}_{L-1}}{K} k_i}_m + \underbrace{\frac{\mathbf{1}^T \bar{a}_{L-1}}{K} \sum_k \frac{e^{z_i^k}}{\sum_j e^{z_j^k}}}_{s_i} \quad (3.16)$$

We end up with an affine description of this indicator $g_i = mk_i + s_i$, with a slope m and the offset s_i , as shown in Figure 3.4. Therefore, if we know these parameters, we can deduce k_i the number of each class, for all class i . The authors propose 3 methods to estimate m and s_i , depending on the assumptions made for the attack:

- LLG : The attacker only has access to the shared gradients and thus to each g_i . In this case, it is assumed that s_i are negligible (which is observed in the case of untrained models) and the slope is estimated using the fact that m is label agnostic: $m \approx \frac{1}{K} \sum_{i \text{ st. } g_i < 0} g_i (1 + \frac{1}{N})$
- LLG* (white-box model): The attacker has access to the model used by the client, meaning that it can be used to generate gradients by passing dummy data in input (e.g. black or random images). The attacker first estimate for each class i the average gradient \bar{g}_i on a dummy batch of B i -labelled samples and then estimate the slope $m \approx \frac{1}{NB} \sum_{i=1}^N \bar{g}_i (1 + \frac{1}{N})$ and the s_i are estimated similarly.

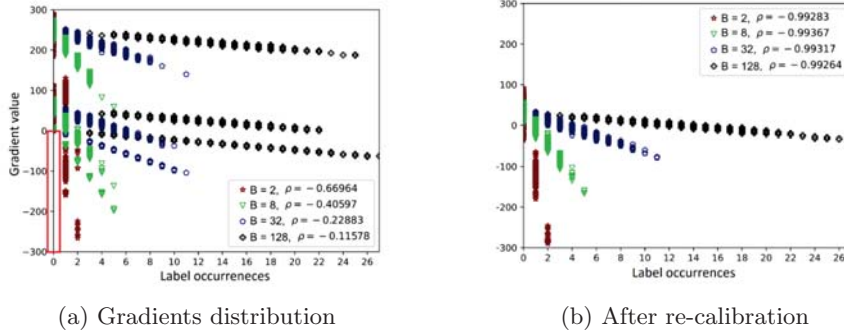


Figure 3.4: Distribution of the gradients g_i (Extracted from [5]). (a) is obtain by randomly initializing a CNN with three convolutional layers and evaluating the gradients (g_i) on MNIST samples across 1000 trials, varying batch sizes $B \in 2, 8, 32, 128$. In (b) the gradients have been re-calibrated by removing s_i , hence a better correlation.

- LLG+ (auxiliary data): In addition to the white-box model, the attacker has an auxiliary dataset that is similar to the one of the client (containing labelled samples of all different classes). This more realistic data is used instead of the dummy data.

These attacks give increasing good results.

3.2.4 iLRG

In 2023, [3] pushed gradient attacks to another level. By leveraging ∇b the gradient of the last layer bias, and making certain approximations, they establish a linear system of equations of the gradients that can be inverted to find the k_i .

First observe that for a biased fully-connected layer $\mathbf{z} = \mathbf{W}^T \mathbf{a} + \mathbf{b}$, $\frac{\partial \mathbf{z}}{\partial \mathbf{b}} = \mathbf{1}$, then the i -th row gradients are simply :

$$\begin{cases} \nabla b_i = \frac{\partial \mathcal{L}}{\partial z_i} \frac{\partial z_i}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial z_i} (= d_i) \\ \nabla W_i = \frac{\partial \mathcal{L}}{\partial z_i} \frac{\partial z_i}{\partial W_i} = \nabla b_i \mathbf{a} \end{cases} \quad (3.17)$$

Therefore for a single input ($K = 1$), if $\exists i$ st. $\nabla b_i \neq 0$ then we can reconstruct the before to last layer output $\mathbf{a} = \nabla W_i \nabla b_i^{-1}$. This can't be done directly for batch data ($K > 1$) so the authors propose the following approximations to estimate the “class-wise embeddings” $\bar{\mathbf{a}}_{\mathbb{K}_i}$.

Approx 1 (Intra-class Uniformity and Concentration of Embedding Distribution). The distribution of embeddings \mathbf{a} is uniform and concentrated over a certain class of samples \mathbb{K}_i in a training batch, such we can replace them with the arithmetic mean of this category, i.e., the geometric center. This approximation holds especially for untrained models.

$$\forall j \in \mathbb{K}_i \quad \mathbf{a}^j \approx \bar{\mathbf{a}}_{\mathbb{K}_i} \quad \Rightarrow \quad \bar{\mathbf{a}}_{\mathbb{K}_i} \approx \nabla_{\mathbb{K}_i} W_i (\nabla_{\mathbb{K}_i} b_i)^{-1} \quad (3.18)$$

Where $\nabla_{\mathbb{K}_i} W_i = \overline{\frac{\partial \mathcal{L}}{\partial W_i}}_{\mathbb{K}_i}$ and $\nabla_{\mathbb{K}_i} b_i = \overline{\frac{\partial \mathcal{L}}{\partial b_i}}_{\mathbb{K}_i}$ and $\overline{(\cdot)}_{\mathbb{K}_i}$ denotes the average over the sub-batch \mathbb{K}_i of only the i -class samples.

Approx 2 (Inter-class Low Entanglement of Gradient Contributions). The batch-averaged gradient row at index i is mainly from i -class samples in a training batch. Namely $\nabla b_i = \frac{k_i}{K} \nabla_{\mathbb{K}_i} b_i$ and $\nabla W_i = \frac{k_i}{K} \nabla_{\mathbb{K}_i} W_i$. This approximation holds especially for untrained models.

Combining these two approximations, they estimate the average class-wise embeddings :

$$\bar{\mathbf{a}}_{\mathbb{K}_i} \approx \nabla W_i \nabla b_i^{-1} \quad (3.19)$$

Approx 3 (Average Probabilities from Average Embeddings). Let $\bar{p}_{\mathbb{K}_j}$ the average post-softmax probability vector from j -class samples. It is approximated to the softmax produced by the j -class average embedding: $\bar{p}_{\mathbb{K}_j} = \overline{\text{Softmax}(\mathbf{W}\mathbf{a} + \mathbf{b})}_{\mathbb{K}_j} = \text{Softmax}(W\bar{\mathbf{a}}_{\mathbb{K}_j} + b)$.

A linear system can finally stem from these approximations. We know that $\nabla_k b_i (= d_i) = p_i - y_i$, so reordering and summing over the K samples gives $\sum_k \nabla_k b_i = \sum_k p_i^k - \sum_k y_i^k$ and therefore:

$$\forall i \in \{1, \dots, N\}, \quad K \nabla b_i = \sum_{j=1}^N k_j \bar{p}_{i\mathbb{K}_j} - k_i \quad (3.20)$$

In addition, we know that $k_1 + k_2 + \dots + k_N = K$ so finally:

$$\begin{cases} \mathbf{1}^T k = K \\ (P - I)k = K \nabla b \end{cases}, \quad \text{with } P = (\bar{p}_{i\mathbb{K}_j})_{(i,j) \in \{1, \dots, N\}^2} \quad (3.21)$$

This is a linear system of the form $Ax = b$ where $x = k$. Since it is not square, it is inverted using the Moore-Penrose pseudo inverse.

This method has the advantage of not necessitating that non negative activation functions are used, but is less effective on trained models since Approx 1 and 2 hold especially for untrained models.

Table 3.1: Assumptions made for each methods (adapted from [1]).

	$K > 1$	Repeating Labels	$a_{L-1} > 0$	Trained Model
iDLG				
BLR				
LLG				
iLRG				

Tables 3.1 and 3.2 compare previous methods based on the conditions required for optimal attack performance, as well as their effectiveness across different datasets and models. These methods give increasing good results. Note (Table 3.2) that the first methods iDLG and BLR can't provide an estimate of the number of instances per class ($iAcc$), and that the last methods LLG and iLRG can also work when FedAvg is used (multiple local epochs §2.2.1).

3.3 Regression

3.3.1 Motivation

Until now, the study of gradient attacks has primarily focused on classification models in the literature, leaving the domain of regression models unexplored. This gap presents a significant opportunity for research, as regression

Table 3.2: Accuracy of the previous methods. Computed in $cAcc(\%) / iAcc(\%)$, where $cAcc$ is class-level accuracy (% of correctly recovered classes) and $iAcc$ instance-level accuracy (% of correctly recovered labels). The “-” means that this metric can’t be computed (for iDLG and BLR). iDLG is adapted to batch recovering by taking the summed gradient (instead of minimum in BLR (3.11)).

	FedSGD			FedAvg
	LeNet-5		ResNet-50	LeNet-5
	CIFAR-100		ImageNet	SVHN
	K=1	K=24	K=32	
DLG	83.3			
iDLG	100	100 / -	89.9 / -	
BLR		100 / -	99.3 / -	
LLG				98 / 98.4 86.3 / 93.4
iLRG		100 / 100		99.8 / 100 99.6 / 99.7
<i>source</i>	[7]	[3]	[6]	[1]

models are widely used in various applications where the labels themselves can be as sensitive as the input data. Indeed, labels are by definition features of interest and thus sensitive. For instance, in medical context, labels could represent critical information such as disease severity which is even more sensitive than personal data such as age, sex or height that are typically input features.

Moreover, while methods such as DLG (§3.1.1) have been shown to effectively reconstruct labels in classification tasks, we observed that these techniques do not perform as well on regression models. As a result, there is a clear need to develop and analyze specialized approaches that address the specific characteristics of regression models.

3.3.2 Our problem setup

In this study, we explored the feasibility of label inference attacks in the context of a regression model being trained in federated learning. More precisely, we considered the following case:

- **Regression model.** We considered a regression model where a single final output neuron with bias b gives the prediction $z \in \mathbb{R}$ with no activation function. This output neuron is fully-connected to the previous layer with weights W such that $z = W^T a + b$, where a is the output of the previous layer. The model predictions \mathbf{z} are compared with the true labels \mathbf{y} using the mean square error (MSE) loss function.
- **FedSGD.** The FL protocol used is the FedSGD algorithm (*cf.* §2.2.1), meaning the shared gradient is the batch averaged gradient of the model parameters over the whole dataset: $\nabla p = \frac{\partial L_{MSE}(\mathbf{z}, \mathbf{y})}{\partial p}$. Therefore, the dataset size is the same as the batch size and will be noted K .
- **Honest-but-curious server.** The attacker is an honest-but-curious server, thus having access to the model and potentially to auxiliary data that is similar to the client data.

Chapter 4

Methods

Inspired by the related work (§3), we explored label reconstruction by leveraging the information contained in the last layer parameters' gradients $(\nabla b, \nabla w)$.

4.1 The $K = 1$ case

Let's first consider the most basic scenario where the client dataset under attack contains only a single sample, i.e., $K = 1$.

4.1.1 An Exact Solution

The MSE loss function computed on this single example is simply $\mathcal{L}(z, y) = (z - y)^2$ where z is the model's prediction and y is the true label. The gradients of the loss with respect to the last layer model parameters are obtained by leveraging the chain rule:

$$\begin{cases} \nabla b = \frac{\partial \mathcal{L}(z, y)}{\partial z} \frac{\partial z}{\partial b} \\ \nabla w = \frac{\partial \mathcal{L}(z, y)}{\partial z} \frac{\partial z}{\partial w} \end{cases} \quad (4.1)$$

where w and b are the weights and bias of the output layer. Reminding the relation of the last fully-connected layer $w^T a + b = z$, note that $w \in \mathbb{R}^M$ and $a \in \mathbb{R}^M$ where M is the length of the before to last layer, and $b \in \mathbb{R}$ just like z and y .

All this partial derivatives can be computed, leading to the gradients expressions:

$$\begin{cases} \nabla b = 2(z - y) \\ \nabla w = 2(z - y)a \end{cases} \quad (4.2)$$

where $a = a_{L-1}$ represents the activations from the layer before the output layer.

By adding the last layer equation $w^T a + b = z$, we end up with a system of 3 equations with 3 unknowns y, z and a . Therefore, assuming that $\nabla b \neq 0$ (it is highly unlikely that $z - y = 0$) it can be solved for y :

$$y = w^T \frac{\nabla w}{\nabla b} + b - \frac{\nabla b}{2} \quad (4.3)$$

This equation provides an exact solution for the label y in the case where the client dataset contains only a single sample.

4.1.2 Generalization to other activation functions

Now assume that the output activation function is non-trivial, i.e. $z = \sigma(w^T a + b)$ where σ is not the identity as above. For example, σ could be the Sigmoid function $\sigma(x) = 1/(1 + \exp(-x))$ if we want the output to be in $(0, 1)$. We can derive the same equations as previously:

$$\begin{cases} z = \sigma(w^T a + b) \\ \nabla b = 2(z - y)\sigma'(w^T a + b) \\ \nabla w = 2(z - y)\sigma'(w^T a + b)a \end{cases} \Rightarrow y = \sigma(w^T \frac{\nabla w}{\nabla b} + b) - \frac{\nabla b}{2\sigma'(w^T a + b)} \quad (4.4)$$

This general expression of y is only valid if σ is differentiable and its derivative does not cancel out. For example, the Sigmoid activation function fulfills this rule but not the ReLU function.

4.2 General case : $K > 1$

Let's now look at the general case when $K > 1$. The MSE loss is now computed on a batch of K predictions-labels pairs. Namely, $\mathcal{L}(Z, Y) = \frac{1}{K} \sum_k (z^k - y^k)^2$ where $Z = (z^k)_{k=1}^K$ and $Y = (y^k)_{k=1}^K$, hence the averaged gradient equations:

$$\begin{cases} \nabla b = \frac{2}{K} \sum_k (z^k - y^k) \\ \nabla w = \frac{2}{K} \sum_k (z^k - y^k) a^k \end{cases} \quad (4.5)$$

This is a much more delicate problem since we now have to reconstruct K labels from the same amount of information. In fact, the gradients are still the same shape as before. Their expression can be rearranged to be written as averaged gradients, e.g. $\nabla b = \frac{1}{K} \sum_k \nabla_k b$ where $\nabla_k b$ is the gradient of b w.r.t. sample k (and similarly for w). Consequently, although we can reconstruct y^k from its sample gradient $(\nabla_k b, \nabla_k w)$ as previously proved, we don't have access to all these sample gradients anymore but only to their mean.

4.2.1 A basic attack: mean y estimation

While the problem stated in 4.5 seems impossible to solve, it is still possible to extract information from the shared gradients. We first developed a basic attack consisting in reconstructing the mean label $\bar{y} = \frac{1}{K} \sum_k y^k$ of the client dataset. In fact, one can rearrange the gradient of b :

$$\nabla b = 2 \left(\frac{1}{K} \sum_k z^k - \frac{1}{K} \sum_k y^k \right) = 2(\bar{z} - \bar{y}) \quad (4.6)$$

Therefore $\bar{y} = \bar{z} - \frac{\nabla b}{2}$ meaning that if we have an estimate of \bar{z} , we can get an estimate of \bar{y} . One simple way of estimating \bar{z} is to pass data through the model and observe the mean prediction. The data used can be more or less

realistic, thus giving a more or less accurate estimate. The attacker can *e.g.* like in §3.2.3 use either dummy data (*e.g.* random, zeros or ones inputs) or some auxiliary data containing samples that are similar to the client’s dataset. The ideal case being when the data used is drawn from the same distribution as the client’s dataset, so the estimate of \bar{z} is unbiased and so is the estimate of \bar{y} .

Algorithm 1 Estimation of \bar{y} using a Dummy Batch

- 1: **Input:** $model, \nabla b$, a dummy batch of size D
- 2: **Output:** Estimate of \bar{y}
- 3: **Step 1:** Generate a dummy batch $X = (x^d)_{1 \leq d \leq D}$ of size D
- 4: **Step 2:** Estimate \bar{z} by computing

$$\hat{z} = \frac{1}{D} \sum_{d=1}^D model(x^d)$$

- 5: **Step 3:** Estimate \bar{y} using the formula

$$\hat{y} = \hat{z} - \frac{\nabla b}{2}$$

- 6: **Return:** \hat{y}
-

4.2.2 Problem transformation

In this section, we present our main method. Instead of reconstructing the labels sample-wise, we transform the problem by making approximations in order to rather aim at reconstructing the label means of multiple subgroups of samples in the client’s dataset.

4.2.2.1 metaclasses

Samples in datasets can often be clustered in groups of samples that are similar. Take the example of a dataset of face images, where the regression task is to predict the age of the person. The samples can *e.g.* thus either be grouped by sex, by skin color, or by having glasses or not. For tabular data *e.g.* a dataset containing different information on employees of a company, one can separate these employees into groups having a common feature *e.g.* the diploma. The first step of our method is to divide the client’s dataset into such groups that we name *metaclasses*.

4.2.2.2 The \bar{y} attack

The goal of this attack is to reconstruct the labels metaclass-wise, *e.g.* leaking the mean age of men and the mean age of women present in the client’s face image dataset.

The gradients equations 4.5 can be written using matrices, giving the fol-

lowing system:

$$\underbrace{\begin{bmatrix} \nabla b \\ | \\ \nabla w \\ | \end{bmatrix}}_{\nabla p} = \frac{2}{K} \underbrace{\begin{bmatrix} 1 & \cdots & 1 \\ | & & | \\ a^1 & \cdots & a^K \\ | & & | \end{bmatrix}}_A \underbrace{\begin{bmatrix} e^1 \\ \vdots \\ e^K \end{bmatrix}}_e \quad (4.7)$$

where $e^k = z^k - y^k$. This is a linear system of equations of the form $\nabla p = Ae$, that we would like to solve for e . This can't be done since the a^k vectors in A are unknown. Therefore, similarly to *Approx 1* in [3] (§3.2.4), we first replace each embedding vector a^k by the mean embedding vector of its metaclass. In the same way, we also replace the errors e^k by their respective metaclass mean. Finally, grouping the samples per metaclass give the transformed system:

$$\nabla p \approx \frac{2}{K} \begin{bmatrix} k_1 & \cdots & k_N \\ | & & | \\ k_1 \bar{a}_1 & \cdots & k_N \bar{a}_N \\ | & & | \end{bmatrix} \begin{bmatrix} \bar{e}_1 \\ \vdots \\ \bar{e}_N \end{bmatrix} \quad (4.8)$$

Where k_i is the number of samples belonging to the metaclass i , N is the number of metaclasses, $\bar{a}_i = \frac{1}{k_i} \sum_{l=1}^{k_i} a^l$ the mean embedding of the metaclass i and $\bar{e}_i = \frac{1}{k_i} \sum_{l=1}^{k_i} e^l$ its mean error.

We now estimate these mean embedding vectors using an auxiliary dataset, *i.e.* we pass the auxiliary inputs X^{aux} through the model and compute the mean before the last layer outputs \bar{a}_i^{aux} per metaclass i . The reduced system $\nabla p \approx \bar{A}\bar{e} \approx \bar{A}^{aux}\bar{e}$ can be solved for the unknown vector \bar{e} (methods to invert this system are detailed in §4.3). Since $\bar{e} = \bar{z} - \bar{y}$, we can finally estimate the mean labels $\bar{y} \approx \bar{z}^{aux} - \bar{e}$ where \bar{z}^{aux} is the auxiliary dataset estimate of \bar{z} .

4.2.3 The k attack

It is also possible to design a similar attack to this time reconstruct $k = [k_1, \dots, k_N]$, the vector containing the number of samples per metaclass. We can also transform system 4.7 like so:

$$\begin{cases} K = \sum_i k_i \\ \nabla b = \frac{2}{K} \sum_{i=1}^N \sum_{k \in \mathbb{K}_i} e^k \\ \nabla w = \frac{2}{K} \sum_{i=1}^N \sum_{k \in \mathbb{K}_i} e^k a^k \end{cases} \Leftrightarrow \begin{cases} K = \sum_i k_i \\ \nabla b = \frac{2}{K} \sum_{i=1}^N k_i \bar{e}_i \\ \nabla w = \frac{2}{K} \sum_{i=1}^N k_i \bar{e}_i \bar{a}_i \end{cases} \quad (4.9)$$

$$\Leftrightarrow \begin{bmatrix} K \\ \nabla b \\ | \\ \nabla w \\ | \end{bmatrix} = \frac{2}{K} \begin{bmatrix} \frac{K}{2} & \cdots & \frac{K}{2} \\ \bar{e}_1 & \cdots & \bar{e}_N \\ | & & | \\ \bar{a}\bar{e}_1 & \cdots & \bar{a}\bar{e}_N \\ | & & | \end{bmatrix} \underbrace{\begin{bmatrix} k_1 \\ \vdots \\ k_N \end{bmatrix}}_k \quad (4.10)$$

This transformation does not require any approximation, it is just a rewriting of the initial system (and adding the sum of k_i). We can now estimate the $\bar{a}\bar{e}_i$

with an auxiliary dataset just like in the previous attack, with the difference that we now need a labelled dataset (X^{aux}, Y^{aux}) to compute the errors e^{aux} and thus the error-weighted mean embeddings $\bar{a}e_i^{aux}$. We can then solve this system $[K, \nabla p]^T = \bar{A}^{aux} k$ for k .

4.2.4 Approximations

This last attack is interesting since it requires different but more realistic approximations. Indeed, the \bar{y} -attack requires that we know the vector k of the client’s dataset, which is a strong assumption since it means that the attacker already have some knowledge on the client’s dataset. This can however be realistic in some cases since the k/K vector can be publicly available, *e.g.* companies publicly sharing their sex ratio. On the other hand, the k -attack doesn’t require any prior knowledge (except for K) on the client’s dataset. In addition, there is no approximation needed to transform the system, whereas the \bar{y} -attack hinged on $\bar{a}e_i \approx \bar{a}_i \bar{e}_i$ for all metaclass i . The only assumption that is stronger for the k -attack is that it needs a labelled dataset.

4.3 Linear Least squares problem

4.3.1 System of linear equations

Both \bar{y} and k attacks hinge on solving a system of linear equations of the form $Ax = b$ where x is the unknown vector, b is the second member, and A is the coefficient matrix that we take of size $m \times n$ only for this section. The system is said to be:

- *underdetermined* if A has fewer rows than columns ($m < n$),
- *square* if A is square ($m = n$),
- *overdetermined* otherwise ($m > n$).

A linear system of equations can either have no solution, a unique solution, or an infinite number of solutions, depending on the shape and rank of A . The easier case is arguably when the system is square and A is full rank (*i.e.* $rank(A) = m = n$). In this case, one can inverse A and get the solution $x = A^{-1}b$, or more efficiently use Gaussian elimination. However, when the system is overdetermined (*resp.* underdetermined), the system generally has no solution (*resp.* infinite number of solutions). It is helpful to see linear systems as a linear combination of the columns of A weighted by the elements of the unknown. Namely, we can write the system under the form

$$x_1 A_1 + \dots + x_n A_n = b \tag{4.11}$$

where $A = [A_1 \dots A_n]$. In this way we can see that the system can’t have a solution if b is not in the span of the columns of A *i.e.* if there is no linear combination x of the columns of A such that $Ax = b$. If the system is underdetermined, there are generally (except in particular if A is very ill-conditioned such that $rank(A) < m$) infinite many ways to write b as a linear combination of $A_1 \dots A_n$ since these column vectors are linearly dependant, hence infinite many solutions. On the other hand, if the system is overdetermined, then they might be no way to write a given b as a linear combination of the columns since b is not necessarily in $span(A_1, \dots, A_n)$ and $span(A_1, \dots, A_n) \subsetneq \mathbb{R}^m$.

4.3.2 Least squares estimate

The systems we are facing in our attacks are mostly overdetermined (more equations than unknowns) since the number of metaclasses N (and thus the number of \bar{y}_i or k_i to reconstruct) is generally smaller than the length of the second member (∇p or $[K, \nabla p]$) which is M (the length of the before to last layer a_{L-1}) plus one or two depending on the attack. Therefore, since there are no exact solution, we aim at finding an approximate solution \hat{x} that best fit the problem. More precisely, one way is to find the *least square estimate* (LSE) *i.e.* a solution to the *least square problem*

$$\hat{x} = \underset{x \in \mathbb{R}^m}{\operatorname{argmin}} \|Ax - b\|^2 \quad (4.12)$$

where $\|\cdot\|$ is the Euclidean norm. The LSE \hat{x} is classically computed by solving the following *normal equation* that satisfies $\frac{\partial}{\partial x} \|Ax - b\|^2 = 0$:

$$A^T A \hat{x} = A^T b \quad (4.13)$$

Providing that A is full column rank, $A^T A$ can be inverted, hence the solution $\hat{x} = (A^T A)^{-1} A^T b$.

4.3.3 The Moore-Penrose inverse

Although this last formula to compute \hat{x} is simple, it is however not ideal for the two main reasons:

- It requires that A is full column rank which is *a priori* not the case when A is underdetermined.
- In cases of near-singularity, where $A^T A$ is close to being non-invertible, the solution can be numerically unstable.

Therefore it is better to use the Moore-Penrose inverse (or pseudoinverse) that gives a more general and numerically more stable solution.

The pseudoinverse can be computed from the singular value decomposition (SVD) of A . For any A there exists a SVD $A = U \Sigma V^T$ where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are unitary matrices ($V^T V = V V^T = I$) and $\Sigma \in \mathbb{R}^{m \times n}$ is rectangular diagonal whose diagonal values are called the singular values. The pseudoinverse of A can be computed leveraging this decomposition by $A^+ = V \Sigma^+ U^T$ where Σ^+ is computed by transposing Σ and taking the inverse of the non zero elements. It can be shown that $\hat{x} = A^+ b$ is the best solution to the least squares problem in the sense that

- In the overdetermined case : \hat{x} is the least square estimate *i.e.* $\hat{x} = \underset{x}{\operatorname{argmin}} \|Ax - b\|$
- If A is invertible : $\hat{x} = A^+ b = A^{-1} b$
- In the underdetermined case : \hat{x} is the (exact) solution with minimum norm *i.e.* $\hat{x} = \underset{x}{\operatorname{argmin}} \|x\|$ s.t. $Ax = b$

Therefore the solution provided by the pseudoinverse is more generalizable since it can be used no matter the shape or rank of A .

4.3.4 Regularization techniques

Although the LSE is the solution that minimizes the error of the system $Ax = b$, it is not necessarily the best solution to the underlying problem. In particular, it is not adapted to the ill-conditioned problems *i.e.* when A is ill-conditioned. In this case, A is very sensitive in the sense that for a small change in x there is a substantial change in Ax , making $Ax = b$ hard to solve. In particular this has the consequence that some small error in b (due to measurement for example) may cause a large error in x .

One solution to overcome this problem is to add a regularization term $R(x)$ to the objective function we are minimizing namely transforming the problem to

$$\min_x \phi(x) = \|Ax - b\|^2 + R(x) \quad (4.14)$$

4.3.4.1 Tikhonov regularization

The simplest regularization is arguably the Tikhonov regularization. Most of the analysis of this section was found in [2].

Definition. This method consists in taking $R(x) = \beta\|Lx\|^2$ thus minimizing the following objective function:

$$\min_x \phi_\beta(x) = \|Ax - b\|^2 + \beta\|Lx\|^2 \quad (4.15)$$

Where L is a matrix defining a semi-norm that can represent *a priori* knowledge on the solution (in practice we only used $L = I$). β is a parameter controlling the balance between the two terms of the objective function. A large beta will lead to a small solution whereas a smaller beta will bring the solution closer to the LSE.

Example. Intuitively, when the problem is ill-conditioned, the LSE can be an absurd solution in the sense that it overfit the noise present in the data. This regularization term will prevent the solution to be absurd by forcing the solution to be “simple” *i.e.* not too large (in the sense of the semi-norm $\|L\cdot\|$). For example let’s take 3 medicines d_1, d_2, d_3 that have a 2-dimensional effect on the body: a good healing effect first dimension and a bad side effects second dimension. We want the combination of drugs that have the best healing effect and no side effect meaning $b = (1, 0)^T$ hence the system

$$\begin{bmatrix} \epsilon_1 & 0 & 1 \\ 1 & -1 & \epsilon_2 \end{bmatrix} x = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (4.16)$$

If there was no noise meaning $\epsilon_1 = \epsilon_2 = 0$, then the least square solution would be $x_0 = (0, 0, 1)$ which is exact. However if there is some small noise $\epsilon_1, \epsilon_2 > 0$ (*e.g.* due to errors in measurement), then the least square solution become the (still exact but absurd) solution $x_\epsilon = (1/\epsilon, 1/\epsilon, 0)$ which is very large since ϵ is very small. By imposing solutions to be not too large, regularization will select a solution close to x_0 which is indeed the solution to the “true” problem (without noise). Therefore the regularization act as an Occam’s razor by selecting a simpler solution even if it is not optimal in term of least squares.

Solution. Just like for the classical least squares problem, we can solve it by differentiating $\phi(x)$ w.r.t. x and forcing this derivative to zero, which gives

the modified normal equation:

$$\frac{\partial \phi(x)}{\partial x} = 0 \Leftrightarrow (A^T A - \beta L^T L)x = A^T b \quad (4.17)$$

This can be showed to be equivalent to solving the following least squares system:

$$\begin{bmatrix} A \\ \sqrt{\beta}L \end{bmatrix} x = \begin{bmatrix} b \\ 0 \end{bmatrix} \quad (4.18)$$

Therefore solving this system with the Moore-Penrose pseudoinverse gives the solution to the regularized problem.

SVD interpretation. Another way to understand Tikhonov regularization is through the SVD. The conditioning of A is measured by its condition number $\kappa(A)$. One way to define $\kappa(A)$ is through the SVD. Let $\sigma_1, \sigma_2, \dots, \sigma_n$ the singular values of A , *i.e.* the diagonal elements of Σ . Then $\sigma_1 > \sigma_2 > \dots > \sigma_n$ and the condition number is $\kappa = \frac{\sigma_1(A)}{\sigma_n(A)} \geq 1$, the higher singular value divided by the smallest. Therefore A is ill-conditioned when it has some small singular values. In short, these small singular values are associated with instabilities and absurd solutions. Therefore a straightforward solution is to apply a cutoff on this singular values *i.e.* solving the least squares system (4.12) where A is replaced by its truncated SVD $\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^T$ where \hat{U} is $m \times r$, $\hat{\Sigma}$ is $r \times r$ and \hat{V} is $n \times r$ where $r \leq \min(m, n)$ is the number of kept singular values. It can be shown that the Tikhonov regularization is equivalent to applying the filter $f_T(\sigma) = \frac{\sigma}{\sigma^2 + \beta^2}$ to singular values therefore penalizing singular values that are $\sigma^2 \ll \beta$.

4.3.4.2 Choice of beta : the L-curve

We know need to find the best β parameter. As discussed previously, this parameter control the trade off between the two terms of the objective function: the misfit $\|A\hat{x} - b\|^2$ and the solution norm $\|\hat{x}\|^2$. We can plot the evolution of the value of this two terms for different values of β on a graph called the *L-curve* due to its classic shape (see Figure 4.1). When β is very small, the solution is typically largest since there is no regulation and the misfit is the lowest (south-east of the L-curve). Increasing β gradually make the solution norm smaller at a price of a larger misfit. The optimal trade-off between misfit and solution norm is intuitively found at the elbow of the L-curve. It can be showed (p. 33 in [2]) that the optimal point satisfies $\text{slope}(L\text{-curve}) = \frac{d \log(\|A\hat{x} - b\|^2)}{d \log(\|\hat{x}\|^2)} = -\frac{n}{m}$ when A is $m \times n$. Other approaches to find optimal β were experimented but we observed that this method gave the best results (in terms of reconstruction error).

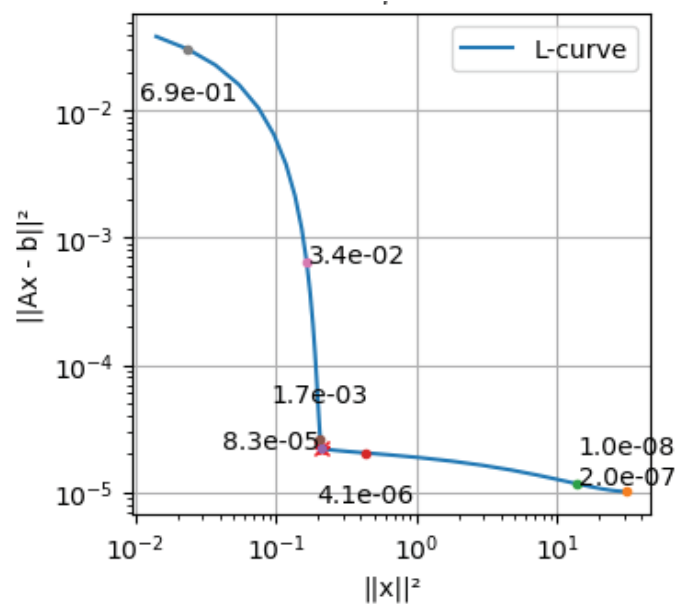


Figure 4.1: An example of L-curve on a given system (one of our attacks). The curve is made by computing the optimal solution \hat{x} minimizing $\phi_\beta(x)$ for different values of β (some are printed on the graph). The x-axis is the solution norm $\|\hat{x}\|^2$, and the y-axis the misfit $\|A\hat{x} - b\|^2$. The red cross shows the elbow of the L, corresponding to the optimal β , found with the method discussed in §4.3.4.2

Chapter 5

Data

5.1 Synthetic dataset

In a research and investigation approach, it is important to be able to easily test different algorithms and hypotheses hence the use of synthetic datasets. Those allow to create specific data structures whose underlying model is controlled and can be modified to observe the reaction of a given algorithm using on this data. For example, one can define the synthetic dataset (X, Y) where $y^k = f(x^k) + \epsilon$ with $\epsilon \sim N(0, \sigma)$ to observe how different intensities of random noise σ affect the training of a ML model.

We use different synthetic datasets to create the FL scenario we are testing our attack on. We first use a simple relation $y^k = \alpha x^k + \beta + \epsilon$ to be sure that a simple MLP would converge. Synthetic datasets also have the advantage to have unlimited samples if needed. We finally use the more complex relation

$$\forall k \quad y^k = \alpha_{i(k)} x^k + \beta_{i(k)} + \epsilon \quad \begin{cases} x^k \sim N(\mu_{i(k)}, \sigma) \\ \epsilon \sim N(0, \sigma_\epsilon) \end{cases} \quad (5.1)$$

where $\alpha(i)$ and $\beta(i)$ depend on the meta-class $i(k)$ of sample k . This has the consequence of making the mean meta-class embeddings \bar{a}_i more different from a class to another, hence an easier problem to solve for our method since A is better condition. This is the synthetic dataset we used for all attacks on synthetic dataset presented in the next chapter.

In practice, we generate a dataset of 6000 samples with 4 meta-classes and an input size of 30, so these values match with the one of salary dataset presented in the next section. The 4 input means are chosen randomly by drawing $\mu_i \sim N_{30}(0, 1) \quad \forall i \in \{1, 2, 3, 4\}$. Similarly, $\alpha_i \sim N_{30}(0, 1)$ and $\beta_i \sim N(0, 1)$, and the noise is set to $\sigma_\epsilon = 1/5$. An instance of this synthetic dataset is shown in the density plot in Figure 5.1.

5.2 Salary dataset

This data comes from this [kaggle](https://www.kaggle.com/datasets/mohithsairamreddy/salary-data/data)¹ dataset. It contains employee data of company, and the task is to predict the salary on other features: age, gender, experience (in years), education (high-school, bachelor, master or phd), and

¹<https://www.kaggle.com/datasets/mohithsairamreddy/salary-data/data>

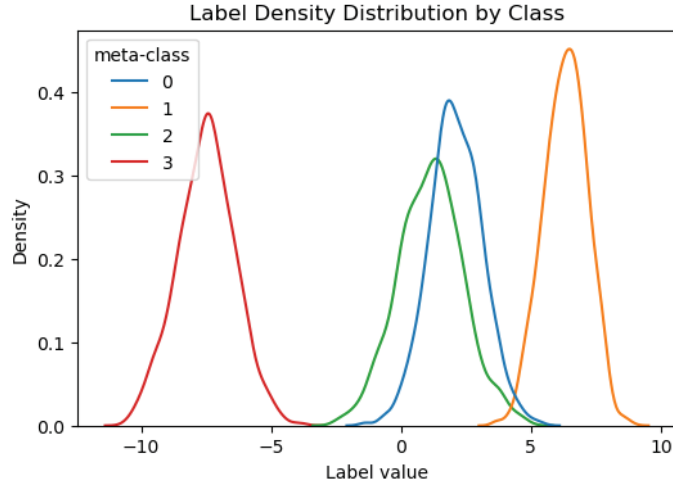


Figure 5.1: Densities of the 4 meta-classes of a synthetic dataset instance.

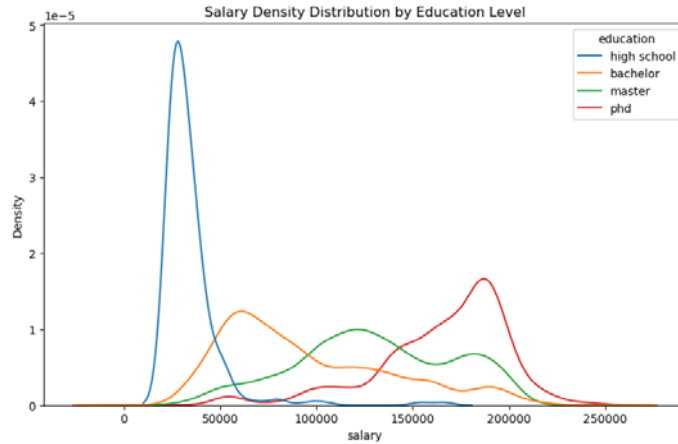


Figure 5.2: Densities of the 4 education meta-classes of the Salary dataset.

job title (*e.g.* “Senior Software Engineer”). It was chosen to experiment our attacks on a real interesting use case, since the salary can be a sensitive data.

The meta-classes are chosen to be directly the 4 education type (high-school, bachelor, master or phd) since this feature is expected to have a substantial importance on the salary prediction, hence we expect the mean class-embeddings \bar{a}_i to be different enough from each other. In practice, we preprocess the data the following way. The salaries range from 25k to 250k (Indian Rupee INR) so we divide them by 10,000 to deal with smaller values. The categorical features are one-hot encoded (with `drop_first`) except for the job title. Since it has 193 distinct values we preprocess this feature the following way to reduce this number. Using regex matching on the job title, we create three new features that contain the role (*e.g.* ‘analyst’, ‘engineer’, ‘director’, etc. 14 values), the field (*e.g.* ‘sales’, ‘marketing’, ‘research’, etc. 13 values), and the seniority (‘junior’, ‘senior’). We choose the most recurrent values for this features. Each features are then one-hot encoded (without `drop_first`). The final input vectors are of length 35.

Chapter 6

Results

In this chapter, we present the performance of different attacks presented in Chapter 4 using different datasets and models and interpret the results.

6.1 Protocol

In order to simulate an attack, we need to simulate the training of a model in a federated context. The models we used were multilayer perceptrons (MLP) of varying inner dimensions. Since the last layer dimension is always 1 for a (non multiple) regression model, we name the model by the list of the hidden layers' lengths except the last one *e.g.* [48, 24] for a MLP that has 3 layers of respectively 48, 24, and 1 neurons. Then we choose the dataset that is either the synthetic or the salary dataset. Both datasets have about 6000 samples, we do a random train/test split of 5500/500 samples. The client's dataset (or "batch") of size K is randomly drawn from the train set at the beginning and kept in memory. It is used at each training round to generate the gradients (using SGD) on the current model that would have been sent to the server. The partition k of the client's batch is also computed and transmitted to the server if necessary. The attack is then executed, using the test set as the auxiliary dataset; we store the error between the reconstructed and true values. Finally, to simulate a training round, the model is trained in a centralized manner on the training set for one epoch using a batch size of 32 and the Adam optimizer. The model test loss is computed on the test set. Then the attack is re-executed on the same client's batch, and so on until the model is trained (the model loss stalls). This whole process constitute one experiment, we repeat 100 experiments with different randomly initialized model parameters and client batch. This allows to have a more robust measure of the attack performance, by its median and mean error.

6.2 $K=1$ reconstruction

When the client's dataset contains a single element ($K = 1$), we showed that we can reconstruct the label of this sample with the explicit formula

$$y = w^T \frac{\nabla w}{\nabla b} + b - \frac{\nabla b}{2}$$

Therefore there is no error in the attack (except potentially numerical error if ∇b is extremely low).

Attack example. Assume a company asks a service provider to build a model that provides a salary scale based on the qualifications of candidates. This service provider might use federated learning to build a model collaboratively with all the company’s employees (each employee being a client possessing one sample which is its own personal data), thus following the recommendations of the CNIL (French Data Protection Authority) on AI. However, if this service provider is honest but curious he will still be able to find the salary of each employee without the employee having shared it directly.

6.3 Basic attack

From now on we consider a client’s dataset size that is more than one. We choose a batch size of $K = 40$ by default for all our experiments. This changes the use case presented in the previous attack. For the salary dataset the classic scenario is now the following: several companies possessing multiple data samples (one per employee) collaboratively train a model predicting the salary of a candidate, and the service provider try to reconstruct data of a company having K employees.

The “basic attack” consisted in reconstructing the mean label of the full client’s dataset by exploiting the relation $\bar{y} = \bar{z} - \frac{\nabla b}{2}$. As discussed above, \bar{z} can be estimated by computing the mean prediction of the model on some data. This data can either be dummy inputs (*e.g.* only zeros, only ones or inputs drawn from a random normal distribution) or auxiliary data that is similar to the client’s data. Figure 6.1 shows the results for this attack in

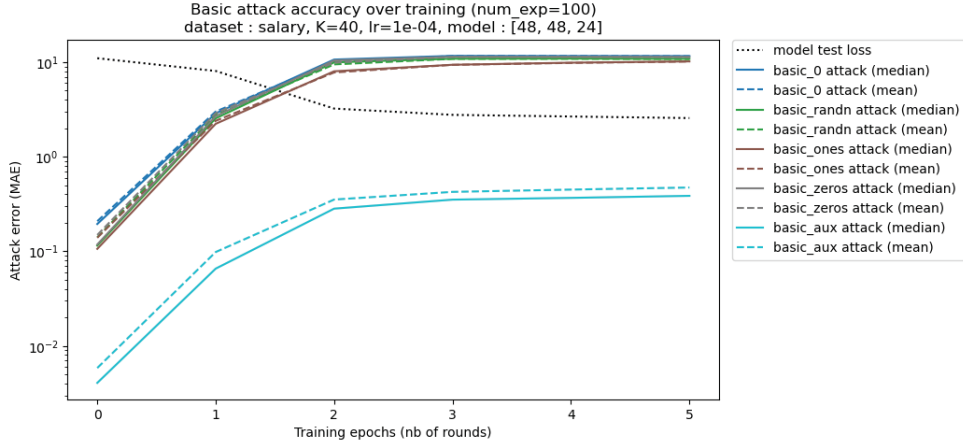


Figure 6.1: Performance of different **basic attacks** depending on how \bar{z} is estimated. **basic_0**: $\bar{z} = 0$, **basic_randn**: random normal dummy inputs, **basic_ones**: ones dummy inputs, **basic_zeros**: zeros dummy inputs, **basic_aux**: auxiliary inputs. The size of client’s batch is $K = 40$. The model is an MLP with inner dimensions [48, 48, 24] trained with Adam with a learning rate of $1e-4$.

the salary dataset case. The results are expressed in absolute error (MAE) between the true mean label of the client batch and the reconstructed mean label. Due to data preprocessing, a MAE of 1 (*resp.* 10^{-2}) corresponds to a 10k ₹ (~120\$) (*resp.* 0.1k ₹ (~1.2\$)) error in salary reconstruction. Alternatively,

it corresponds to 4.4% (*resp.* 0.04%) of the salaries’ total range (salaries range from 25k ₹ (~ 300 \$) to 250k ₹ (~ 3000 \$)). We observe that as expected, the attack using auxiliary data instead of dummy data gives much better results. Since the attack error is the error on \hat{z} ($MAE(\hat{y}, \bar{y}) = |\hat{y} - \bar{y}| = |\hat{z} - \bar{z}|$), this is because the auxiliary dataset provides a way better estimate of \bar{z} than dummy data. When the model is trained for example (at *training epoch* = 5 on Figure 6.1), this error is small for the auxiliary data since $\bar{z}^{aux} \approx \bar{z} \approx \bar{y} \approx 10^1$ whereas for the dummy data, $\bar{z} \approx 10^{-1}$ hence the 10^1 error.

We also observe that for all the variants, the attack is the more performing at the first step of the training *i.e.* at the first round of communication when the model is untrained. Then the error of the attack drastically increases. For the attacks based on dummy data (`basic_randn`, `basic_zeros` and `basic_ones`), this is because their estimation of \bar{z} remain within the same order of magnitude $\hat{z} \approx 10^{-1}$ during the training, whereas the true value of \bar{z} goes from 2×10^{-1} to 10^1 (this last fact can be seen on Figure 6.1 on the `basic_0` attack curve since $\hat{z} = |\hat{z} - 0| = |\hat{z} - \frac{\nabla b}{2} - (0 - \frac{\nabla b}{2})| = MAE_{\text{basic}_0}$). Therefore, the error $|\hat{z}^{dummy\ data} - \bar{z}|$ is of the same order of magnitude as \bar{z} , hence the increase during the training.

6.4 \bar{y} attack

This attack consisted in grouping the dataset samples in “meta-classes” and reconstructing the mean label \bar{y} for each of these meta-classes. Since attacks on labels in the case of regression have never been explored in the current literature, we can’t compare the performance of our attacks to a state of the art baseline. Therefore we compare the reconstruction error of our attack to a **naive attack** that could be done with the same hypotheses. The requirements for this attack were:

- Honest but curious:
 - knowing the gradients ($\nabla b, \nabla w$)
 - knowing the model
- Additional:
 - knowing the partition vector k of the client’s dataset
 - having an auxiliary dataset of (only) inputs X^{aux} and k^{aux}

The naive approach we choose in this context is to take the mean model prediction per meta-classes as the reconstructed label *i.e.* $\bar{y}_i = \overline{z^{aux}_i} = \overline{model(X_{class=i}^{aux})}$. This attack is especially inefficient when used on the first rounds since an untrained model outputs random predictions, and it gets better as the model learns. We also looked at the error of the previous “basic attack” that predicts $\bar{y}_i = \bar{y}_{basic\ atk}$ no matter the meta-class i . This attack is more efficient than the previous naive attack at the very first rounds, but has a constant error over the training.

We compared two attacks to these naive attack baselines: the method using directly the Moore-Penrose pseudoinverse (MP) to solve the system, and the one using Tikhonov regularization (T) in addition. We used the mean absolute error (MAE) to measure the attack error, $MAE(\bar{y}, \hat{y}) = \frac{1}{N} \sum_{i=1}^N |\bar{y}_i - \hat{y}_i|$.

6.4.1 Synthetic dataset

On the synthetic dataset (defined in §5.1), the attack gives relatively good results. Figure 6.2 show similar performances for the two attacks, and both errors are largely below the naive attack error (between 3 and 20 times smaller). Both errors decrease during the training of the model. This might be due to the fact that when the model learns, the \bar{a}_i^{aux} become different from one meta-class to an other and thus the system contains a more clear information; it is then easier to decompose the gradients into a weighted sum of the $\bar{a}_i \bar{e}_i$. So far

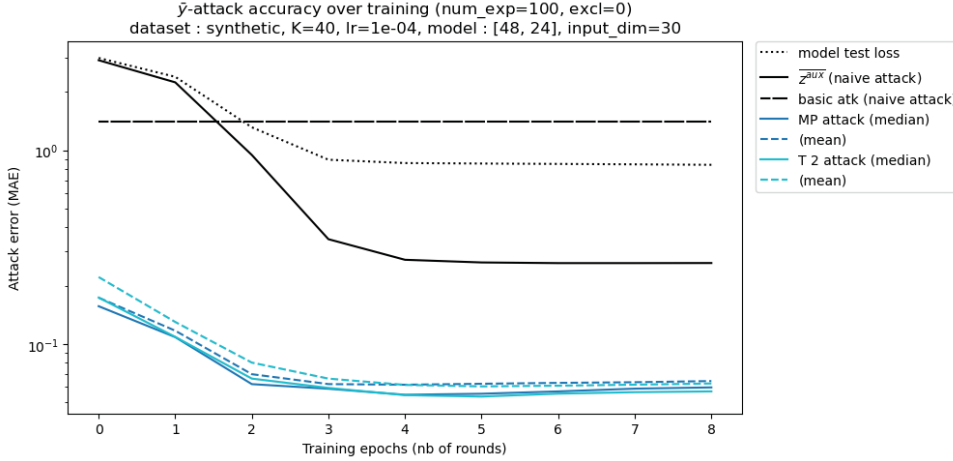


Figure 6.2: Moore-Penrose (MP) and Tikhonov (T) attacks on the synthetic dataset.

we don’t see the benefits of regularization, but it can have a substantial effect on the error when the problem is more ill-conditioned. For example when we decrease the last layer’s length of the model, we observed the problem to be harder to solve. Therefore doing the same attacks in such “ill” conditions gives Figure 6.3 that shows two interesting things:

1. Both attacks have the same median but the attack with regularization (T) has a substantially lower mean.
2. Without regularization (MP), the mean error is highly correlated with the condition number of the A matrix.

This is due to the fact that when the model is ill-conditioned, the solution of the non-regularized problem (MP) can be excessively large and therefore pulling the mean. For example in Figure 6.3 one MP-solution error (over the 100) reached $3.6e2$ at the 5th round, which mean a solution \hat{y} of the same magnitude since the true labels have a magnitude of $|\bar{y}| \approx 1e0$ (in this case). Such solution is absurdly large. The regularization make all these absurd solution vanish by imposing a penalty on the norm of the \hat{y} .

6.4.2 Salary dataset

The same attacks results on the salary dataset are presented Figure 6.4. We observe this time a substantial benefit of the regularization on the reconstruction error of the mean labels per meta-class. The attack with Tikhonov regularization gives an error that is constantly 2 times smaller than the error

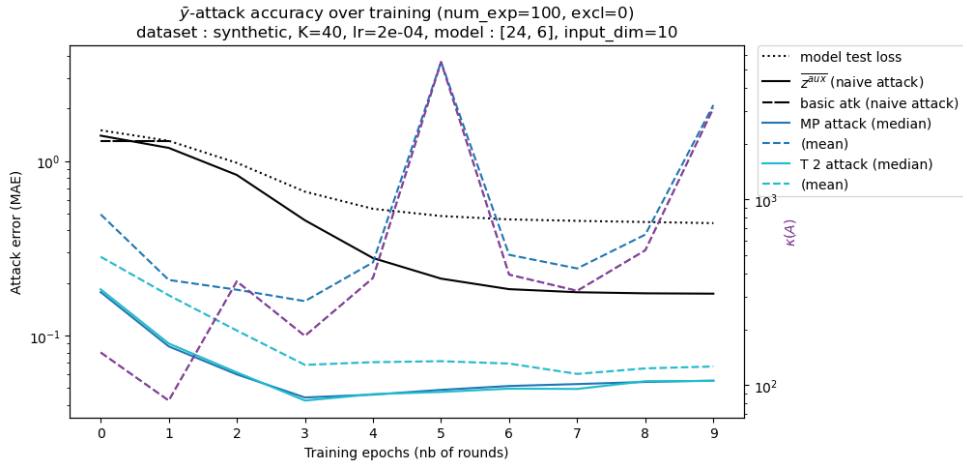


Figure 6.3: Moore-Penrose (MP) and Tikhonov (T) attacks on an ill-condition problem. The model is smaller than before as well as the input size of the synthetic data. For readability we only show the first basic attacks errors since it is constant.

of the Moore-Penrose attack without regularization. This allows the Tikhonov attack performance to stand out from the naive attack whereas it is not the case for the Moore-Penrose attack. The Tikhonov attack reaches a MAE of 1.5 which corresponds to an error of 15k ₹ (~179 \$) in the salary real unit (indian rupee), which corresponds to 6.6% of the salary total range (salaries range from 25k ₹ (~300 \$) to 250k ₹ (~3000 \$)).

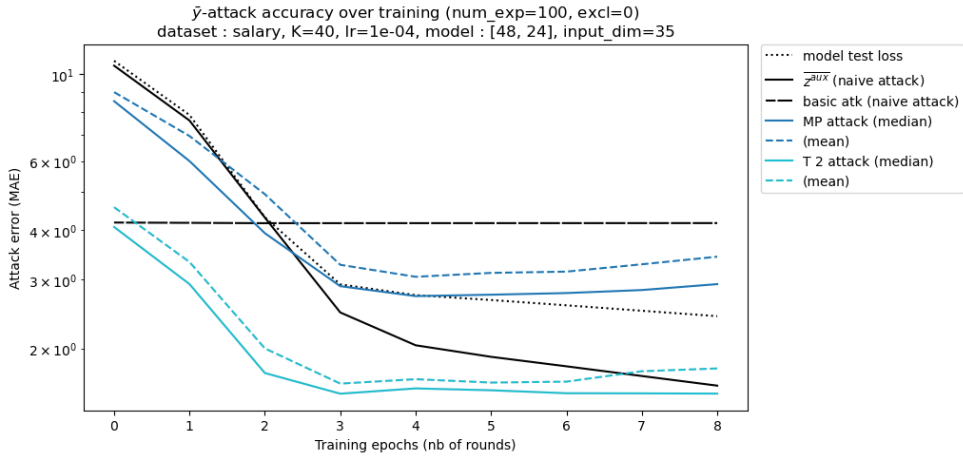


Figure 6.4: Moore-Penrose (MP) and Tikhonov (T) attacks on the salary dataset.

6.5 k attack

We now analyze the results of the k -attack whose goal is to reconstruct the partition vector $k = [k_1, k_2, \dots, k_N]$ where k_i is the number of client's samples belonging to class i .

Remind the assumptions for this attack:

- Honest but curious:

- knowing the gradients ($\nabla b, \nabla w$)
- knowing the model
- Additional:
 - knowing the dataset size K
 - having an auxiliary labeled dataset (X^{aux}, Y^{aux}) and k^{aux}

In this case, a naive attack is simply the random prediction of a k vector that sums to K . This is the baseline we chose to compare our attacks with.

ASR. To measure attack performance, we use the Attack Success Rate (ASR). This metric is used in classification attacks [5], it is the ratio of the number of correctly matched meta-class samples over the total number of meta-class matches (K). For example, if $k = [1, 21, 10, 8]$ (typical value for the salary dataset) and $\hat{k} = [10, 10, 10, 10]$, then $ASR(k, \hat{k}) = \frac{1+10+10+8}{40} = 0.725$. We also used MAE to have a more intuitive interpretation of the error *e.g.* $MAE(k, \hat{k}) = 5.5$ in the previous example.

Figure 6.5 shows the performance of MP and T attacks on the synthetic dataset. Contrary to the \bar{y} -attack, the k -attack performs best at the first rounds of the training and then collapse as the model is trained. It also seems correlated with the condition number. The attack with regularization (T) has once again the best performances, its median ASR (*resp.* MAE) is always above (*resp.* below) the one of the MP and random attacks.

Similar results were obtained for the salary dataset as shown Figure 6.6. But this time it seems that the attack performances aren't correlated to the condition number. The median MAE of the T attack error at the first round is 2.5.

6.6 Robustness experiments

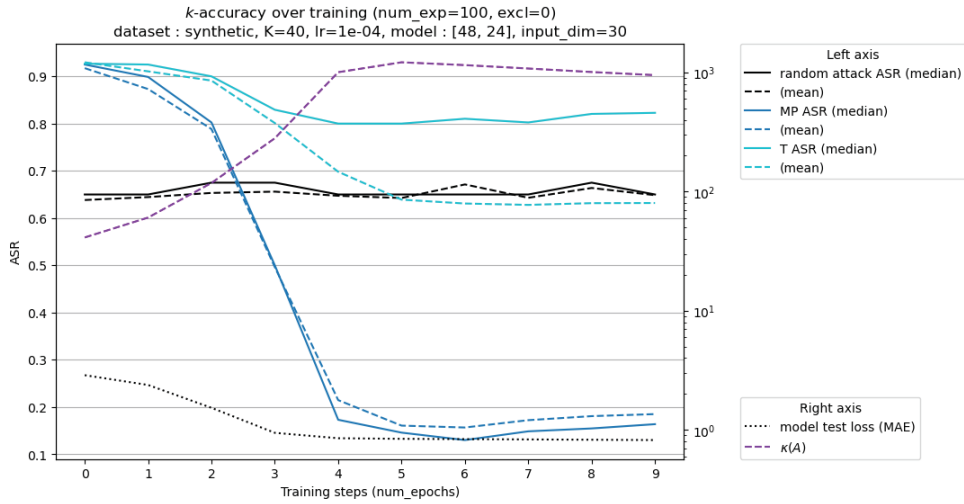
Let's test the robustness of these attacks with respect to the variation of different parameters.

6.6.1 Batch size K

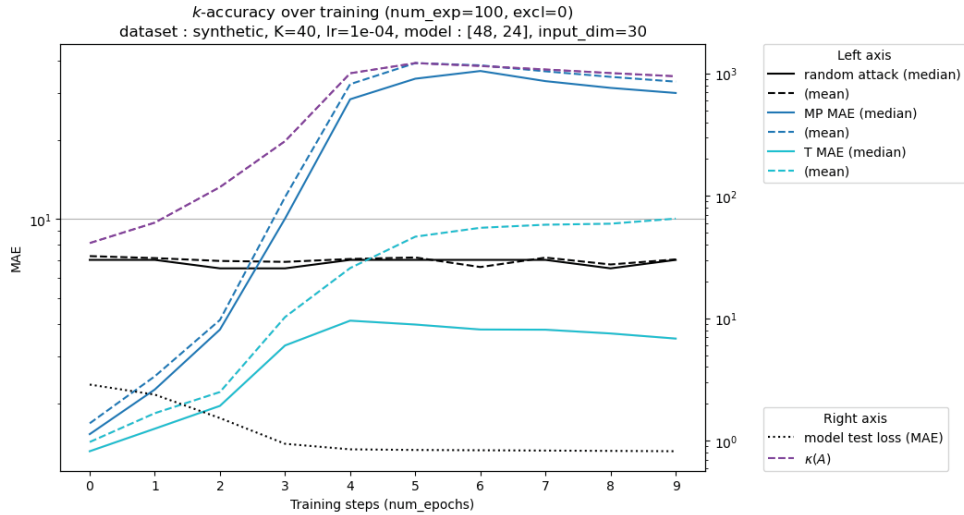
The client's batch size is a parameter that is expected to change from a FL scenario to an other. Therefore we test our y -attack on the salary dataset for different values of K : 10, 20, 80, and 160. These attacks give similar results for these values as shown in Figure 6.7, but we note a slight improvement as K increases for the final value of the T attack. However this tiny improvement is caught up by the improvement of the naive attack. Indeed since the client's batch is larger, its mean salaries per class are less likely to be far from the full dataset means (law of large number in a sense) that \bar{z}^{aux} is estimating.

6.6.2 Last layer size M

We also explore the attack performance under different last layer's lengths M . This parameter was expected to have an impact since it determines the number of rows of the system $\nabla p = Ax$. Figure 6.8 suggests that the attack accuracy decreases with M . In particular, the mean error (dotted lines)



(a) ASR



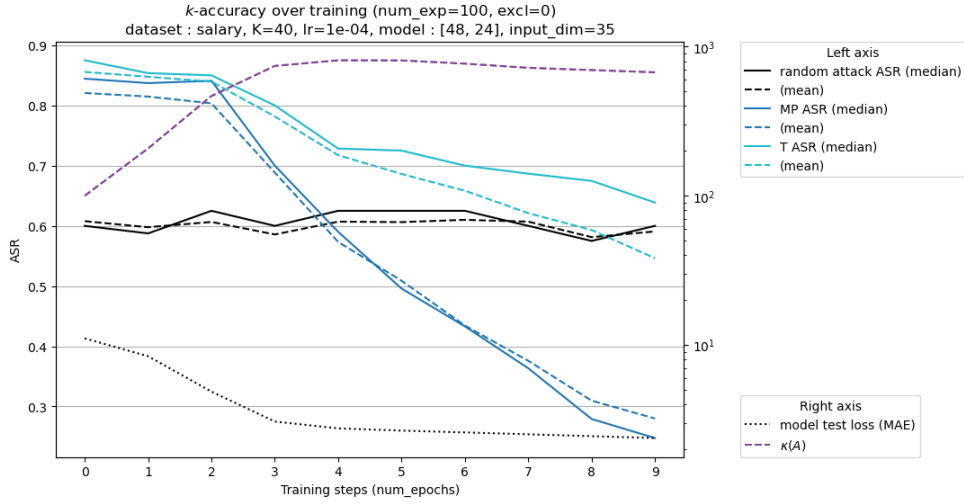
(b) MAE

Figure 6.5: Moore-Penrose (MP) and Tikhonov (T) k -attacks on the **synthetic** dataset. Attack performances are measured with two different metrics: ASR (a) and MAE (b). The client’s dataset size is $K = 40$ and the model is an MLP [48,24].

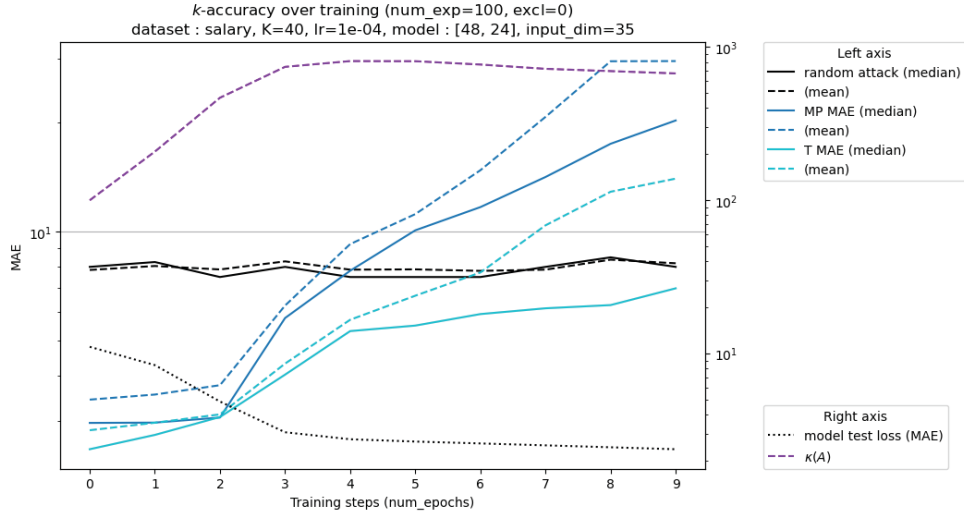
increases as the last layer size is reduced. The MP attack (without regularization) seems more sensible to this parameter than the T attack (with regularization), making this last attack more robust *w.r.t.* different M values.

6.6.3 Auxiliary data fit

One of our main hypothesis is that the server has access to an auxiliary dataset that is similar to the client’s dataset. This was achieved in our experiments by drawing (without replacement) the client’s dataset (X, Y) and the auxiliary dataset (X^{aux}, Y^{aux}) from the same original dataset. To challenge this hypothesis and see how our attacks behave when this approximation is not fulfilled, we voluntary transform the auxiliary dataset of the server by



(a) ASR



(b) MAE

Figure 6.6: Moore-Penrose (MP) and Tikhonov (T) k -attacks on the **salary** dataset. Attack performances are measured with two different metrics: ASR (a) and MAE (b). The client’s dataset size is $K = 40$ and the model is an MLP [48,24].

applying:

$$\begin{cases} X^{aux} = X^{aux} + \epsilon & \text{where } \epsilon \sim N_{M \times N}(0, \sigma_\epsilon) & \text{for the } y\text{-attack} \\ Y^{aux} = Y^{aux} \times \gamma & & \text{for the } k\text{-attack} \end{cases} \quad (6.1)$$

Figure 6.9 shows that adding a Gaussian noise with a standard deviation up to 1 has a small effect on the attack performance. Figure 6.10 shows that a shift in the label values has a direct effect on the first rounds attacks’ ASR that is non negligible even for a 10% shift.

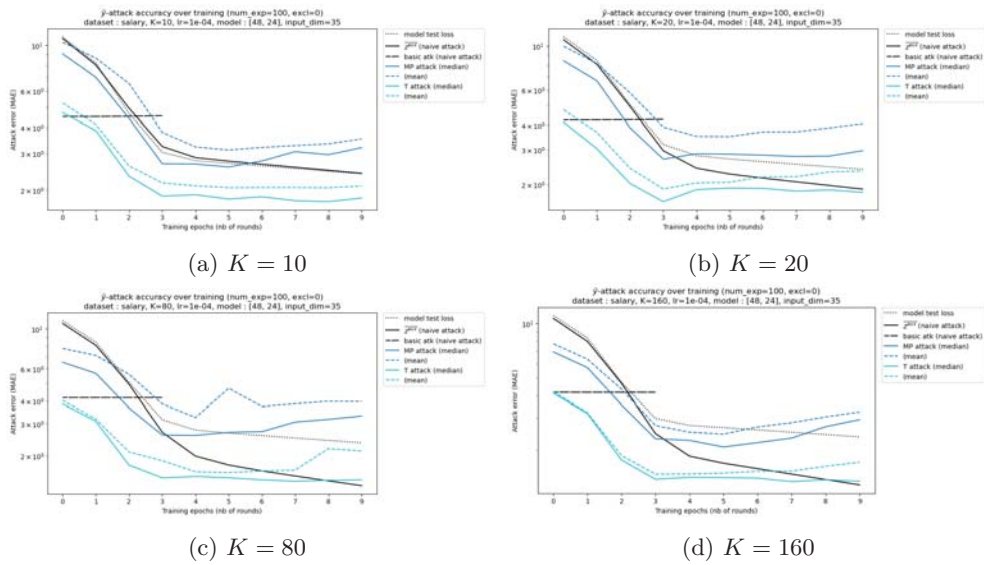


Figure 6.7: Moore-Penrose (MP) and Tikhonov (T) y -attacks on the salary dataset. The model is an MLP [48,24].

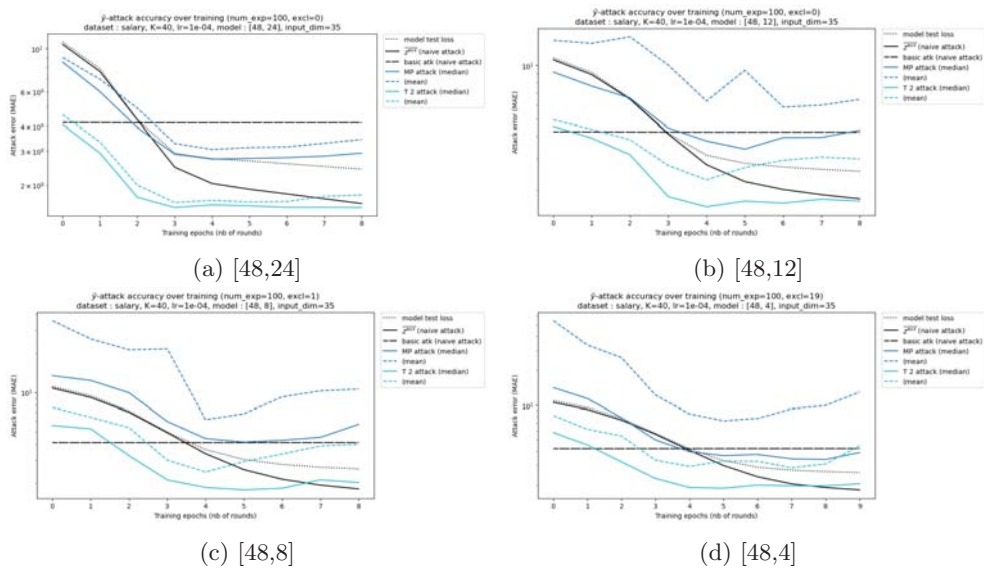


Figure 6.8: Moore-Penrose (MP) and Tikhonov (T) y -attacks on the salary dataset for MLPs of different last layer sizes (notation explained in §6.1). The client's dataset size is $K = 40$.

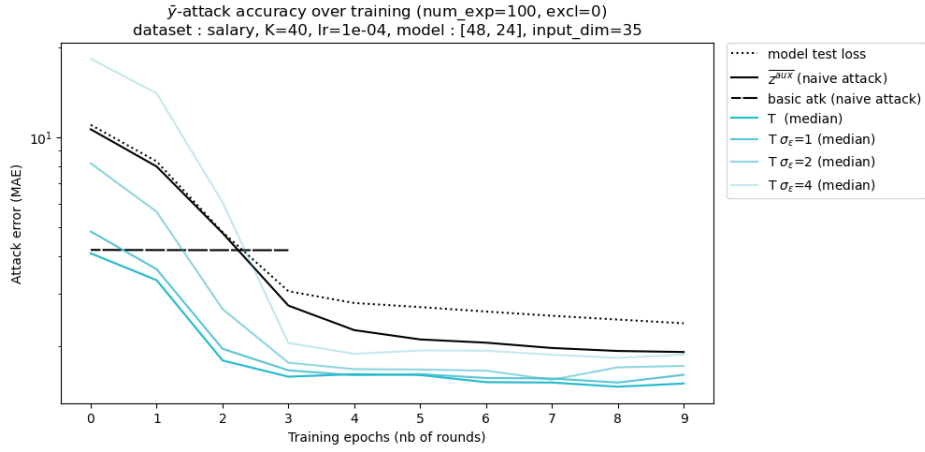


Figure 6.9: Tikhonov y -attack (T) for different values of σ_ϵ . The auxiliary inputs X^{aux} are transformed according to 6.1. Salary dataset, $K = 40$, MLP [48,24].

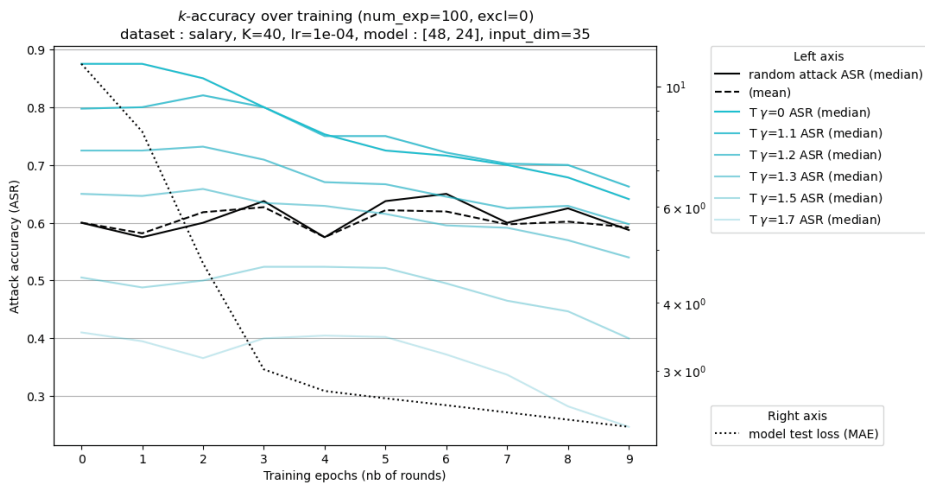


Figure 6.10: Tikhonov k -attack (T) for different values of γ . The auxiliary labels Y^{aux} are transformed according to 6.1. Salary dataset, $K = 40$, MLP [48,24].

Chapter 7

Conclusion

In this thesis we explored gradient attacks on labels in the context of a regression model trained under a federated learning framework. We explored different reconstruction attacks. First a simple but exact reconstruction of y when $K = 1$. When the client’s dataset contains more than one element which is usually the case, it is much more difficult to reconstruct instance-wise information from the gradients. We show through the “basic attack” that it is possible to infer with relatively good accuracy the mean label of the batch, especially when the model is untrained. We then introduced the meta-classes and proposed an attack reconstructing the mean label per meta-class. This attack performs relatively well on the synthetic dataset, but its accuracy is mitigated on the salary dataset while still better than our baselines. Our last attack aimed at reconstructing the partition vector k of the client’s batch with respect to the meta-classes rather than the labels. This attack has the advantage of not necessitating a strong assumption, compared to the previous attack that required the server to know the k vector.

Further research. It might be interesting to assess the external validity of this last two methods on other datasets. For example the human face dataset (where the regression task is to predict the age) may expose more complex relationships between inputs, outputs and meta-classes (gender, color skin, etc.), and therefore altering or improving the attack performances. Since these attacks require strong assumptions that may not be satisfied in a real scenario, it would be interesting try to reduce their need. One could for example use the result of the k -attack for the y -attack. Finally, to increase the attack accuracy, more sophisticated regularization terms may be explored, to more precisely penalise the “absurd” solutions leveraging the *a priori* knowledge of the server on the labels. For example, meta-classes means are likely to be distributed around the mean label (that can be estimated with the basic attack); also the k vector must be non-negative.

Bibliography

- [1] Huancheng Chen and Haris Vikalo. Recovering labels from local updates in federated learning. *arXiv preprint arXiv:2405.00955*, 2024.
- [2] Eldad Haber. *Numerical strategies for the solution of inverse problems*. PhD thesis, University of British Columbia, 1997.
- [3] Kailang Ma, Yu Sun, Jian Cui, Dawei Li, Zhenyu Guan, and Jianwei Liu. Instance-wise batch label restoration via gradients in federated learning. In *The Eleventh International Conference on Learning Representations*, 2023.
- [4] H. B. McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *International Conference on Artificial Intelligence and Statistics*, 2016.
- [5] Aidmar Wainakh, Fabrizio Ventola, Till Müßig, Jens Keim, Carlos Garcia Cordero, Ephraim Zimmer, Tim Grube, Kristian Kersting, and Max Mühlhäuser. User-level label leakage from gradients in federated learning. *Proceedings on Privacy Enhancing Technologies*, pages 227–244, 04 2022.
- [6] H. Yin, A. Mallya, A. Vahdat, J. M. Alvarez, J. Kautz, and P. Molchanov. See through gradients: Image batch recovery via gradinversion. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 16332–16341, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.
- [7] Bo Zhao, Konda Reddy Mopuri, and Hakan Bilen. idlg: Improved deep leakage from gradients. *arXiv preprint arXiv:2001.02610*, 2020.
- [8] Ligeng Zhu, Zhijian Liu, and Song Han. *Deep leakage from gradients*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Appendix A

$$\begin{aligned}
\nabla W &= \frac{1}{K} \sum_k \begin{pmatrix} + & + & + & + & + & + & - \\ + & + & + & + & + & + & - \\ + & + & + & - & + & + & + \\ + & + & + & - & + & + & + \\ + & + & + & - & + & + & + \\ + & + & + & - & + & + & + \\ + & + & + & - & + & + & + \end{pmatrix} \begin{pmatrix} + & - & + & + & + & + & + \\ + & - & + & + & + & + & + \\ + & + & + & + & + & + & + \\ + & + & + & + & + & + & + \\ + & + & + & + & + & + & + \\ + & + & + & + & + & + & + \end{pmatrix} \begin{pmatrix} + & - \\ + & - \\ + & - \\ + & - \\ + & - \\ + & - \end{pmatrix} \\
&\quad \begin{matrix} \swarrow K \\ \searrow \end{matrix} \quad \begin{matrix} \leftarrow N \\ \rightarrow \end{matrix} \quad \begin{matrix} \leftarrow M \\ \rightarrow \end{matrix} \\
\nabla W &= \begin{pmatrix} + & - & + & + & + & + & - \\ + & - & + & - & + & + & + \\ + & - & + & - & + & + & - \\ + & - & + & + & + & + & - \\ + & - & + & + & + & + & - \\ + & + & + & - & + & + & - \\ + & - & + & - & + & + & - \end{pmatrix} \\
\min_m \nabla W &= \begin{pmatrix} + & - & + & - & + & + & - \end{pmatrix} \\
\hat{c} &= \begin{pmatrix} 2 & 7 & 4 \end{pmatrix}
\end{aligned}$$

Figure A.1: Batch Label Restoration algorithm of [6] (§3.2.2) on a batch of 3 samples ($K = 3$). The $+$ signs show that the negative sign can be lost during the averaging (in blue), this is why taking the minimum of each column is more robust than the sum.

