École Polytechnique Fédérale de Lausanne

Secure and private machine learning using homomorphic encryption

by Florian Singer

Master Thesis

Prof. Serge Vaudenay
EPFL Supervisor

Nicolas Casademont
Company Supervisor

Ketl
33 Rue des Bains
1205 Genève, Switzerland

September 4, 2022

Acknowledgments

I would like to thank the whole team at Ketl for their support during my internship, in particular Nicolas Casademont, James McGill and Samuel Halff for their valuable input on the written report and oral presentation, as well as Prof. Serge Vaudenay at EPFL for supervising this project.

Lausanne, September 4, 2022 Florian Singer

Abstract

With the ever increasing use of machine learning models processing millions of data points, techniques for protecting this data have emerged. The use of homomorphic encryption is promising but still suffers from strong performance issues and relatively low adoption outside of the academia. First, we formalize the threats related to machine learning and then attempt to solve the problems of secure model training and private inference using homomomorphic encryption. In particular, we use logistic regression models for the training problem, and neural networks for inference. We implement the solutions using two popular homomorphic encryption libraries and analyze the results in terms of runtime, memory usage and accuracy, with the perspective of a real-world deployement. We find that heavy optimizations or security and privacy sacrifices need to be made for many use-cases, but we also explore trade-offs that can be acceptable, such as shifting some of the computations to the client.

Contents

Chapter 1

Introduction

The boom of artificial intelligence and machine learning seen in the past decade has had significant impact on societies, with the rise of recommender systems, image recognition software and large-scale advertisement platforms, to name a few. These advances can be beneficial, but they come with many risks and challenges that are often disregarded for the sake of short-term profit.

One of the main challenge is related to the sensitivity of the data used in these systems. As many different actors are usually participating in the training and usage of a machine learning (ML) model, the data providers need to place a high level of trust in the other actors.

This trust can be achieved through legal bindings, but only technological safeguards can provide real guarantees that the data will not be misused.

One of these safeguards is to hide the data with encryption. However, the mainstream cryptography of today only protects the data while in transit or at rest, and it cannot be used when performing operations on it. This limitation is fortunately lifted in a particular subset of cryptographic schemes, called homomorphic encryption (HE) schemes. These have the advantage of allowing some operations, primarily additions and multiplications, while keeping the numbers fully encrypted.

The downside of this approach is that it comes with severe performance and precision limitations, and is thus difficult to apply in the context of machine learning where computations are already gluttonous in resources. Nevertheless, this has been an active research field that has seen promising advances in the past decade.

The goal of this project is to identify, implement, evaluate and possibly improve the state-of-the-art solutions of homomorphically encrypted machine learning. It aims to determine if such solutions are practical for real-world systems, notably by exploring the trade-offs between security and performance.

In chapter 2, we will introduce two different types of ML models, namely logistic regression and neural networks, and the basics of homomorphic encryption. Then, we will formalize in chapter 3 the threats that exist in ML processes, and describe how they can be mitigated using homomorphic encryption. More specifically, we will see how HE can protect data during both the training phase and the inference phase, and what it entails to transform an ML algorithm to a homomorphically-friendly version. The implementation details and challenges are given in chapter 4. Finally, we present and analyze the different performance metrics in chapter 5.

Chapter 2

Background

This chapter introduces the main concepts that form the basis of this work, namely machine learning (2.1) and homomorphic encryption (2.2). It will be useful for understanding the subsequent chapters.

2.1 Machine Learning

The main ML models of interest for this thesis are logistic regression models, neural networks and convolutional neural networks.

2.1.1 Logistic regression

Logistic regression (LR) is a simple and widely used model for supervised classification. Given a set of n training points $\mathbf{x}_i \in \mathbb{R}^d$ and the corresponding label $y_i \in \{-1, 1\}$ among two classes, the training phase consists of learning a vector of weights $\mathbf{w} \in \mathbb{R}^{d+1}$ minimizing the loss function:

$$\mathscr{L}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \log(1 + \exp(-\mathbf{z}_i^T \mathbf{w})) \tag{2.1}$$

where $\mathbf{z}_i = y_i \cdot (1, \mathbf{x}_i)$ for $i = 1, ..., n$.

A common method used to minimize the loss function is the gradient descent algorithm. It iteratively updates the weights vector by following the direction opposite to the gradient of the

function:

$$\begin{aligned}\mathbf{w}^{t+1} &= \mathbf{w}^t - \gamma \nabla \mathscr{L}(\mathbf{w}) \\ &= \mathbf{w}^t + \frac{\gamma}{n} \sum_{i=1}^{n} \sigma(-\mathbf{z}_i^T \mathbf{w}^t) \cdot \mathbf{z}_i \end{aligned} \tag{2.2}$$

where $\gamma > 0$ is the learning rate and σ is the sigmoid function (or logistic function) defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

Classifying a new data point $\mathbf{x}$ can be done as follows:

$$y = \begin{cases} 1, & \text{if } (1, \mathbf{x}) \cdot \mathbf{w} > 0 \\ -1, & \text{otherwise} \end{cases} \tag{2.4}$$

2.1.2 Neural networks

Neural networks (NN) are defined as a succession of *layers*, each consisting of operations on a set of input neurons, which are typically real numbers.

Basic networks make use of *fully connected* (or *linear*) layers, which are defined as such for an input vector $\mathbf{x} \in \mathbb{R}^d$, a weights matrix $W \in \mathbb{R}^{k \times d}$ and a bias vector $\mathbf{b} \in \mathbb{R}^k$:

$$h^{\text{linear}}(\mathbf{x}) = W\mathbf{x} + \mathbf{b} \tag{2.5}$$

Usually, a non-linear *activation function* is applied to the output in order to learn more complex decision boundaries. The most common is $\mathrm{ReLU}(x) = \max(0, x)$, but the sigmoid (Equation 2.3) or the hyperbolic tangent function are also often used.

The training phase is made of two main steps. First, a data point (or a batch of data points) is evaluated successively through the layers, in what we call the forward pass. Then, we perform a gradient descent by computing the gradient of the loss function at each neuron and by updating the weights, starting from the end of the network. This is called the backward pass.

Once the model is trained, it can be used to classify new data points. For this operation, only the forward pass is required. Each output neuron typically corresponds to a class, and the output values represent the likelihood that a point is part of this class.

2.1.3 Convolutional neural networks

In many practical cases, particularly for image recognition, the input dimensions can be quite big. This is a problem, because a linear layer connects every input neuron with every output neuron, which results in a quadratic number of operations. In order to improve performance and better exploit the spatiality properties of pixels in images, we can limit the connections of a pixel to its neighborhood. This amounts to applying a convolution.

For an image $x \in \mathbb{R}^{c \times h \times w}$ with c channels, a height of h and a width of w pixels, a 2D convolutional layer is defined as follows for each output channel i:

$$h_i^{\text{conv2d}}(x) = b_i + \sum_{k=1}^{c} w_{i,k} \star x_k \tag{2.6}$$

where b_i is the bias, $w_i \in \mathbb{R}^{c \times k_1 \times k_2}$ is the kernel and $\star$ is the 2D cross-correlation operation.

Another important type of layer is the *pooling* layer, that will aggregate a neighborhood of neurons into one value. The type of aggregate is usually a *max* operation or an *average*.

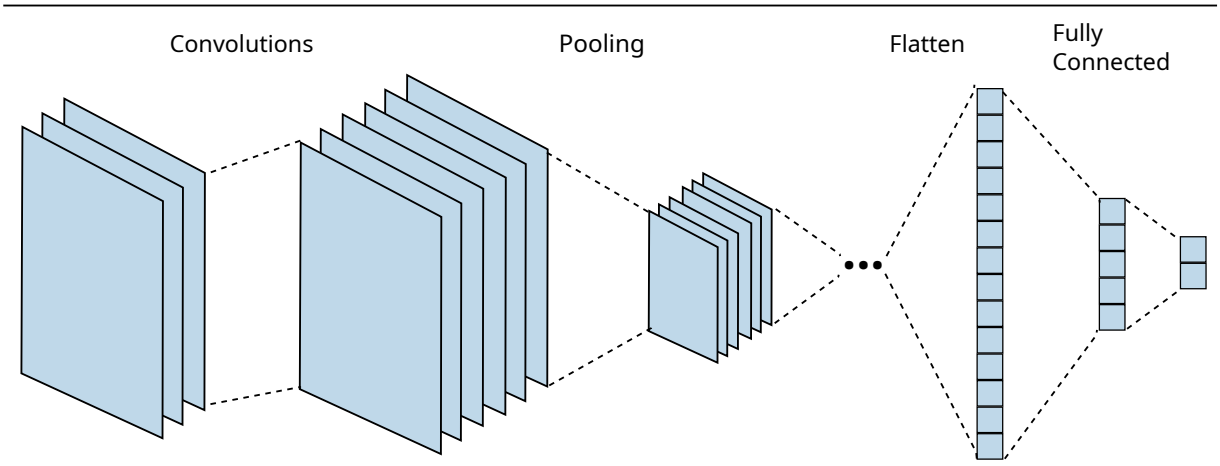

Figure 2.1: Traditional CNN architecture.

As illustrated in Figure 2.1, the traditional architecture of a convolutional neural network (CNN) starts with an alternate of convolutions, activation functions and pooling layers. Each feature learned is encoded into a channel, and the convolutions will progressively increase the number of channels. The role of the pooling layers is to reduce the size of these channels. At some point, the input is flatten into one dimension and feeded to a sequence of fully connected layers.

This covers basic CNN architectures for supervised classification, but more advanced tools are often included in modern networks, such as *batch normalization*, *dropout*, or *skip connections*. There are also different types of architectures, such as *recurrent neural networks* that are used for

temporal data. We will not go into such details, but support for these advanced methods could be done as future work.

2.2 Homomorphic encryption

The basic idea of homomorphic encryption is to allow computations on encrypted data, which is not possible with most modern and commonly used schemes. This concept has been introduced in 1978 by the authors of the RSA cryptosystem [1]. The first schemes supported only one type of operation, such as homomorphic additions with the Paillier cryptosystem (1999) [2].

More formally, for a scheme supporting homomorphic additions on two elements x and y of a set, there should be an operation $\oplus$ as well as encryption and decryption operations Enc and Dec respectively, such that :

$$Dec(Enc(x) \oplus Enc(y)) = x + y \tag{2.7}$$

The same principle holds for homomorphic multiplications, with a different operation $\otimes$.

The main revolution of the field was due to Gentry [3] in 2009, when he presented a blueprint for constructing fully homomorphic encryption schemes. Fully homomorphic schemes support two types of primitive operations, typically additions and multiplications, and can evaluate arithmetic circuits of arbitrary depth thanks to a *bootstrapping* procedure.

From this blueprint were born many cryptosystems that are used today, such as BGV [4], BFV [5], or CKKS [6, 7]. The former two work with vectors of integers, and the latter deals with vectors of complex numbers. Here is an overview of the basic operations available with CKKS:

- `KeyGen(params)`: Generate a public key and a private key.
- `Encode(x)`: Encode a vector of complex numbers into a plaintext, which is a polynomial in the ring $R = \mathbb{Z}[X]/(X^N + 1)$.
- `Decode(x)`: Decode a plaintext back into a vector.
- `Encrypt(x, pub_key)`: Using the public key, encrypt a plaintext into an element of $R^2_{q_L}$ where $R_{q_L} = R/q_L R$ is the set of polynomials from R with coefficients modulo q_L.
- `Decrypt(x, sec_key)`: Decrypt a ciphertext using the private key.
- `Add(x, y)`: Pairwise addition of two ciphertexts, or one ciphertext and one plaintext.
- `Multiply(x, y)`: Pairwise multiplication of two ciphertexts/plaintext.
- `Rotate(x, k, rot_key)`: Permute the elements in a ciphertext by shifting them by k positions to the left.

- `Relinearize(x, rel_key)`: After a multiplication, the resulting ciphertext will be in $R^3_{q_l}$ and will continue to grow after each multiplication. Relinearization is used to bring it back to $R^2_{q_l}$ and facilitate decryption. It requires a relinearization key, sometimes also called an evaluation key.
- `Rescale(x)`: Rounding operation performed to maintain the size of a ciphertext after a multiplication. It computes $\lfloor \frac{q_{l'}}{q_l} \cdot x \rceil \in R^2_{q_{l'}}$ for a ciphertext $x \in R^2_{q_l}$.
- `Bootstrap(x)`: Refresh a ciphertext once it reached its maximum number of operations, by putting it back in $R^2_{q_L}$. Details are available in [8] and [9].

During the initialization phase, some parameters need to be set:

- The *ring degree* N determines the size of the vectors that we encode. It should be a power of two. A higher degree also provides more security.
- The *moduli chain* is a sequence of integers $\{q_0, ..., q_L\}$. The number L should be at least the multiplicative depth of the function to be computed, because one element is usually "consumed" for each rescaling operation, and one more is needed during decryption.
- The *scaling factor* Δ of the plaintext, influencing the precision of the result. It is used to discretize real numbers during encoding, and should not be confused with the rescaling operation.

These parameters need to be carefully tuned based on the problem at hand. Other parameters, such as the *error distribution* or the *key distribution*, are often set to the same standard value.

The basic concept of HE, while already powerful, is limited to two entities. It is not possible to mix data encrypted with different keys. In a scenario where multiple actors want to collaboratively use their data without revealing it to each other, it would be unsafe for all of them to use the same private key. To solve this problem, two approaches have been proposed.

The first is *Multi-Key Homomorphic Encryption* (MKHE). It was first introduced by López-Alt et al. [10] in 2012, as an extension of the NTRU HE scheme. Then, Chen et al. [11] adapted this idea for bootstrappable schemes such as BFV and CKKS. In this setting, any party can encrypt data and join the pool of computations. There is no need for an initialization step between parties, which means that the participants do not need to be known in advance. It is only during the decryption process that every party involved has to participate.

Multiparty Homomorphic Encryption (MHE) is the second approach suggested by Mouchet et al. [12]. In this case, the key generation process must be done collaboratively with every participant in order to create a common public key. This provides less flexibility, but the advantage lies in efficiency. As opposed to MKHE, the size of ciphertexts, rotation keys, and relinearization keys do not grow with the number of parties.

Chapter 3

Design

Now that the main mathematical and cryptographic concepts have been introduced, we focus on the problems to be solved. In this chapter, we formalize the privacy and security threats that need to be mitigated. We also describe our solutions to some of these issues from a high-level perspective, and explain the important design decisions made for these solutions.

3.1 Threat model

Instead of focusing directly on specific issues, it is useful to draw an overview of the threats that a machine learning system can face. This will help us understand how homomorphic encryption can be used to mitigate some threats, and how these mitigations can be bypassed.

3.1.1 Data-flow diagram

In order to better visualize how a machine learning system looks like, Figure 3.1 attempts to model the data flow in a general case. This diagram could be refined for more concrete applications. For instance, the model provider and the data collector may be the same entity, which would remove a trust boundary. Another gray area is the feature extraction process that can happen in different ways: it could be done directly on the client and the data collector, or it could rely on an external service controlled by the model provider.

Thus, in this case, the goal of this diagram is to be a tool for identifying the threats, rather than an exhaustive reference.

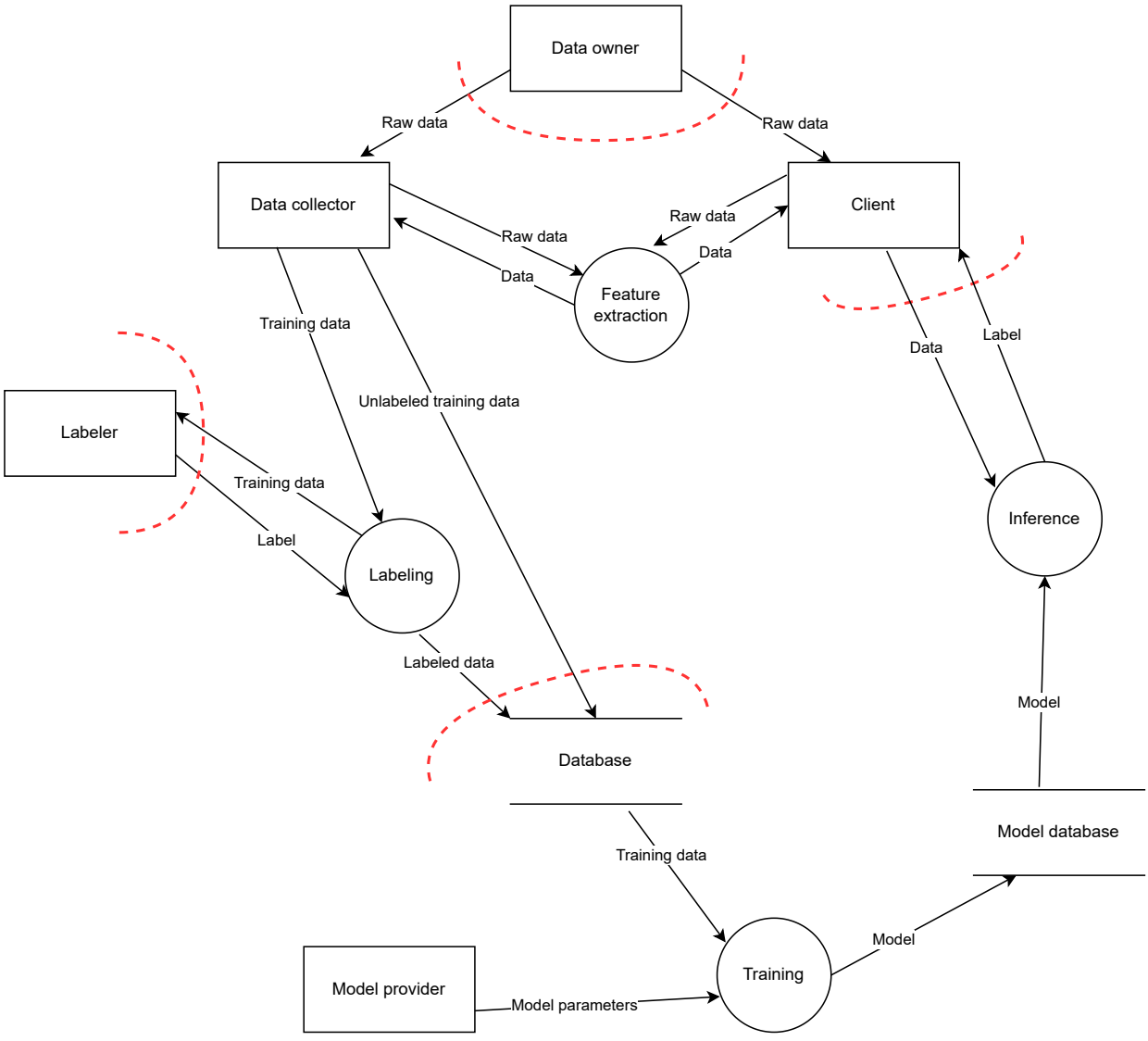

Figure 3.1: Dataflow diagram for a general ML system. Rectangles represent entities, circles represent processes, arrows stand for data flows, and trust boundaries are illustrated with dashed lines. It is around these trust boundaries that we expect many threats to appear, because it often implies a communication between two entities.

3.1.2 Threats on ML systems

From surveys of the ML security literature [13, 14], we can identify many common types of threats.

Poisoning attacks

In many cases, an attacker wants to influence the training process by injecting or modifying the input data. Let us take the example of a classification model based on textual data from documents. To carry out a poisoning attack, an adversary could create spam documents with uncommon words. This will reduce the utility of the system by replacing useful words in the feature vector. If the system lets adversaries define the labels, the model can also be poisoned with incorrect labels. This type of attack can be detected with the help of another model specialized in anomaly detection.

Evasion/impersonation attacks

For this type, the attack vector is the same but it involves carefully crafted inputs in order to produce a specific result. These attacks are often carried out to evade spam filters or to impersonate someone in a biometric recognition system.

Data ordering attacks

In the same vein, data ordering attacks aim to reduce the utility or even bias the model by putting some data points first during training [15]. This could be used to introduce racist or sexist biases, for instance. Thus, it is important to shuffle the training data properly.

Model extraction attacks

If the attacker has unlimited access to the prediction results, it is possible to reconstruct and steal an approximation of the model. The typical adversary carrying out this attack could be a competitor unwilling to dedicate the resources necessary for training a model from scratch.

Inference attacks

This family contains multiple subcategories, but the general goal is to extract some knowledge from the training dataset. It can either be general or target a single individual.

A common type is **membership inference**, where the adversary tries to determine if a known sample was part of the training data. In some cases, being part of a dataset is a sensitive information in itself, such as for criminal or medical records. The intuition behind this attack is the following. If a model returns a high confidence value for a sample, this sample is likely to have been encountered during training. This is due to the degree of overfitting that usually appears in practice when training ML models.

Sometimes, only part of a sample is known and the goal is to find a sensitive attribute. This is called an **attribute inference** or **model inversion attack**. For instance, an attacker may want to infer the medical conditions of a person, using some known attributes (age, gender, weight) and the result of a classifier trained on this data.

To be successful, the attacker sometimes requires data poisoning [16], or needs access to additional information, such as confidence levels or auxiliary datasets [17, 18].

Finally, **property inference attacks** aim to leak general features of a dataset, even if these are not explicitly encoded and are not part of the learning task. This is problematic because the data subject is usually not aware of the type of information that can be derived.

Other threats

Referring back to the dataflow diagram in Figure 3.1, we see that most of these attacks occur on the trust boundary around the database, and the one around the client. There is also a privacy risk when the training data is transmitted to the labeler, which could be a third party mandated by Amazon Mechanical Turk or other similar services.

As we will see in details in the following sections, there are two other threats that are of particular interest to us, one during the training process and one during the inference process.

3.2 Secure training

In addition to the threats that we identified in subsection 3.1.2, another one is that external actors, such as hackers or cloud providers, could observe the training process and exfiltrate the data. A potential defense mechanism is to train the model on homomorphically encrypted data.

In our case, we assume that the whole training data matrix is encrypted with the same key. The holder of this key is both the model owner and the data owner. The data should be encrypted in a secure enclave before sending it to the training service.

If the data comes from external data owners and should not be visible to the model owner, we can design a protocol using multi-party homomorphic encryption. However, this feature is

not available in many HE libraries.

If we distinguish the client (data and model owner) and the server (computing power provider), the steps of an encrypted training procedure can be described as follows:

Client	Server
1. Generate keys	
2. Send relinearization and rotation keys	
3. Encrypt training data	
4. Send encrypted data	
	5. Training
	6. Send back encrypted weights
7. Decrypt weights	

As training can be a fairly complex and resource-intensive process, it is wise to start with a simple classification model. Logistic regression models are simple and have been extensively studied as part of the 2017 iDASH secure genome analysis competition. The winning solution has been proposed by Kim et al. [19]. This will be the basis of our implementation.

However, it is necessary to make a few adjustments to remove some limitations. The input matrix is padded with zeros such that the number of columns is a power of two. This step is necessary for efficiently summing the elements of each row in logarithmic time. Then, the matrix is packed row by row into one or multiple ciphertexts. This method supports an arbitrary number of data points but requires that the number of features is lower than the number of slots, which is a power of two usually between $8‘192$ and $32‘768$.

For the training step, a regularization term has also been added. Apart from its common usage in ML for avoiding overfitting, it has in this case the added advantage of improving the precision of the computations. This is because the learned coefficients are kept small, and the sigmoid approximation that we use is only precise for a certain interval around 0.

The pseudocode for the encryption and training algorithm is given in the appendix (Algorithm 1). It basically translates Equation 2.2 into an algorithm containing only homomorphic operations on vectors. For maximal efficiency, we need to be careful to minimize the total number of operations, as well as the number of chained multiplications (multiplicative depth).

The inference step described in Equation 2.4 can also be translated to a homomorphic version in order to have a completely encrypted pipeline, but this scenario will be described in the following section with a more complex model.

3.3 Private inference

3.3.1 Overview

In the case of an inference service provided by a model owner to a client, another threat is that the client's data can be abused. Even with a privacy policy, the client has no technical guarantee that this data will not be shared, stolen, or used for different purposes. As opposed to many attacks mentioned in subsection 3.1.2, this threat is independent of the ML model and involves only the data. Homomorphic encryption aims to reduce this risk, thus improving the overall trust in the system.

There are two approaches to this problem. In both cases, we assume that a model has been trained beforehand.

In the most common approach, the client encrypts the data and sends it to the inference service, which learns nothing about the data and the inference result.

In the second approach, the client computes the inference by themselves with the trained model. Unfortunately, few services will accept to send the model in cleartext, in fear of it being stolen. However, the weights of the model can be encrypted and sent to the client. This requires an additional round-trip of communications for the decryption step, which can reveal the inference result to the service if no precautions are made. In addition, the architecture of the model must be revealed to the client, which incurs the risk of it being stolen and retrained from scratch by a competitor. Due to these limitations, we will focus on the first approach, but the runtime estimates should not differ too much between the two, because the algorithms are quite similar.

This time, if we consider the client to be the data owner and the server to be the model owner, the steps are very similar to section 3.2:

Client	Server
1. Generate keys	
2. Send relinearization and rotation keys	
3. Encrypt data	
4. Send encrypted data	
	5. Inference
	6. Send back encrypted label
7. Decrypt label	

Note that the inference step can also be delegated to an external cloud provider without risks of compromising the data. In addition, the first two steps only need to be done once if the client makes multiple inference requests over time.

Because predictions require much less resources than training, we can look at more complex models. As neural networks are extremely popular nowadays, a deep learning framework compatible with encrypted data would have many use-cases.

3.3.2 Data packing

This problem, especially for the special case of convolutional neural networks, has already been widely studied. One of the first solution was proposed by Dowlin et al. [20] in 2016. One particularity of their approach lies in how they encode the data into vectors. They pack the n pixels of an input image into n different ciphertexts. This is very efficient when batching multiple images (by putting the pixels of the first image in the first slots, the pixels of the second image in the second slots, etc) but it results in a lot of wasted space if only one image needs to be evaluated. From now on, we will call this method *pixelwise packing*.

An alternative is to pack the pixels of an image one after the others in the same ciphertext. This results in a more efficient use of space when doing few inferences. For simplicity, we assume that all the pixels fit into a single ciphertext. We also separate each channel into its own ciphertext. For this reason, we call this method *channelwise packing*.

Both methods have their own trade-offs that highly depend on the architecture of the network. They are evaluated side by side in chapter 5. From the theoretical complexity, we expect pixelwise packing to be best when kernels and fully connected layers are small.

In terms of memory, pixelwise packing may produce more ciphertexts, but it has the advantage of not needing any rotation key, which can take a few gigabytes when the ring dimension is large.

For our framework to be flexible on the architecture of the model, the same type of packing is kept throughout the layers. However, it would be possible to optimize a specific architecture by switching between multiple packings, as it is done for LoLa in [21]. The authors describe other packing methods optimized for certain tasks.

The pseudocode for a convolutional layer with channelwise packing is given as an example in Algorithm 2 of the appendix.

3.3.3 Hybrid approach

As will be shown in chapter 5, a fully encrypted inference on a deep network can still be very time-consuming. This is mainly due to the total depth of the network and to the first convolutional layers that extract features from an image.

These first layers are often generic and sometimes rely on publicly available pre-trained

weights. From the perspective of the model owner, making these layers public can be acceptable because most of the knowledge derived from the data is concentrated in the last classification layers.

Thus, a hybrid solution alternating between cleartext and encrypted parts could provide an important performance improvement at the expense of a reasonably small security sacrifice (the increased chance of a model extraction attack). Although more computing power is required from the client, they keep the same guarantees in terms of data privacy. It should however be assessed whether the divulgation of these weights represents a privacy risk from the perspective of the training data owner. Indeed, it could facilitate some of the attacks mentioned in subsection 3.1.2, such as membership inference or model inversion attacks.

The architecture of a hybrid model is illustrated in Figure 3.2. In this case, the client computes all the convolutions and pooling layers, then flattens and encrypts the result. The server only has to evaluate the fully connected layers on encrypted data.

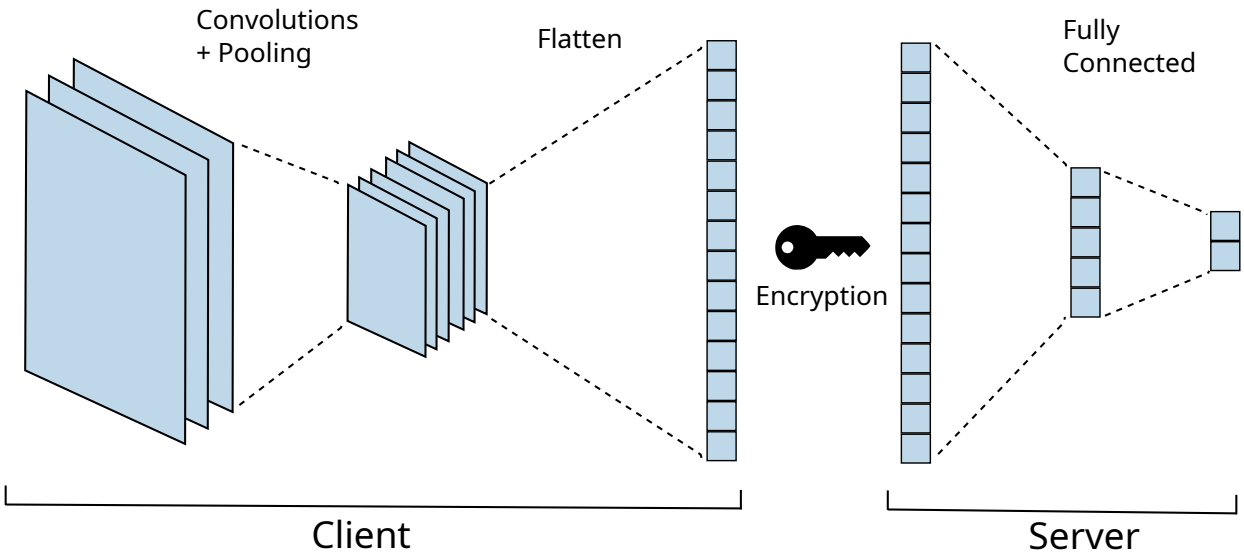

Figure 3.2: Hybrid CNN inference, the first part is done on the client in cleartext and the last layers are computed on the server.

Another advantage of this approach is that the first layers are not restricted in the range of operations, because they are computed in cleartext. This means that no approximations are necessary for non-polynomial functions, and that popular layer types like ReLU or max-pooling are available.

Chapter 4

Implementation

In the previous chapter, we described the different security threats that occur in ML systems. We identified that data can be at risk in both the training and the inference steps, and that homomorphic encryption is a way of protecting this data. We started with a high-level description of the solutions, before focusing on how to apply these principles for specific ML models, namely logistic regression and neural networks.

This chapter covers the lower-level aspects of the solutions. It presents the libraries used and the architecture of the implementation. It also highlights the challenges faced during development, as a way to introduce the limitations.

4.1 Backends

There are multiple HE libraries providing the CKKS scheme. Most of them are implemented in `C++`, such as SEAL [22], PALISADE [23] or HEAAN [24], but Lattigo [25] is implemented in Golang. A more complete survey and comparison of the different libraries can be found in [26].

Each of these libraries has its own particularities, as there is not yet a complete standard for homomomorphic encryption (although the work has been started in [27]). In the next subsections, we will introduce the ones that have been used in this project.

Despite these differences, we implemented a higher-level abstraction to unify the usage of these libraries, and to provide a more intuitive API for non-expert programmers. Thanks to this, the ML algorithms can be implemented independently of the backend. The set of available primitives is slightly reduced, because the relinearization and rescaling operations are performed automatically when necessary. There are also utility functions to export the keys and ciphertexts into a `base64` encoded JSON object. This gives an example of how clients and servers can share these objects in practice.

One of the requirements of the project was to code in Python. This means that our implementation needs bindings to use these C++ libraries.

4.1.1 PALISADE

PALISADE is a widely used open-source library. It provides incomplete bindings for Python, but we can extend them as needed.

In its basic usage, the library creates a context from a set of parameters and stores the keys needed for the evaluation (such as rotation keys). It is slightly higher-level than other libraries by default, e.g., it rescales ciphertexts automatically. It also provides a wide set of utility functions, such as polynomial evaluation. Surprisingly, this function is not optimized to minimize the multiplicative depth, which is critical in our case. Thus, we still need to implement our polynomial functions manually. Finally, multi-party HE is supported, but there is no bootstrapping procedure.

4.1.2 SEAL

SEAL, developed by Microsoft, is one of the most popular libraries. A complete set of Python bindings is already available [28].

It offers finer grained initialization parameters and a lower-level API (e.g., we need to set the moduli size at each level, and rescaling needs to be done manually). However, it does not offer bootstrapping, nor a multi-party version.

4.1.3 Plaintext

When dealing with ciphertexts, it is extremely useful to have a cleartext pipeline that can be inspected for easier debugging. It can also be used to evaluate the cost of translating an algorithm to a homomorphic-encryption-friendly version, independently of the cost of encryption. For this backend, we used `numpy` vectors to represent the ciphertexts.

4.2 Logistic regression

The API for the logistic regression implementation is inspired by other ML libraries. It consists of a class providing a `train` and a `predict` function, along with their encrypted counterparts. Separate functions are available for encrypting and decrypting the data. This modularization is important for any real setting where the client and the server are different applications.

Complete pipelines are also provided as an example and for the evaluation. It shows which parts should be run on the client, which parts should be run on the server, and which objects must be shared.

Since we do not have access to bootstrapping with SEAL and PALISADE, the number of gradient descent operations is limited. To overcome this, it would be possible to send the intermediate weights to the client and re-encrypt them. However, this way the client learns some information on the weights.

4.3 Neural networks

Similarly to logistic regression, a class wraps the encryption, prediction and decryption functions. It also translates layers defined with the `pytorch` library into an encrypted version. The following layers are supported, for both the *pixelwise* and the *channelwise* packing methods:

- Fully connected layer of any size
- 2D convolution with zero-padding
- 2D average pooling with stride
- Approximate sigmoid activation function (from [19])
  $f(x) = 0.5 - 1.20096 \cdot (x/8) + 0.81562 \cdot (x/8)^3$
- Approximate $\tanh$ activation function (from [29])
  $f(x) = -0.00163574303018748 \cdot x^3 + 0.249476365628036 \cdot x$
- Square activation function $f(x) = x^2$

The pipelines show an example of a client-server separation, as well as a non-encrypted training procedure for learning and storing the model weights.

4.4 Challenges

Many problems arise when implementing or using these algorithms. First, the limited number of operations available with homomomorphic encryption forces us to find approximations or workarounds for some functions. This is the case for the sigmoid or $\tanh$ functions. These approximations are usually only precise in a certain range around zero. If the computations are not kept in this interval, they risk to overflow or become extremely biased. Some other operations such as conditionals are also very expensive and complex to implement [30].

Then, we are in practice limited in the number of operations that we can chain. Even if CKKS with bootstrapping does not theoretically have this limitation, the imprecision of each computation, inherent to the scheme, will make the error grow continuously. Also, the bootstrapping operation is not available in all libraries and still takes several seconds at best [9].

An important consideration is to be aware of the security level of the encryption. The security level is commonly defined as a number of bits. It estimates roughly the number of operations that an attacker would have to do on a ciphertext to learn some information about the private key or the data. The Homomorphic Encryption Standard [27] suggests values of the ring dimension N and the coefficient modulus q that achieve a security level of 128, 192, and 256 bits. A security of 128 bits is often the minimum for future-proof applications, but 80 bits can also be considered sufficient given the computing power of current computers. Thus, many researchers in the field of homomomorphic encryption use a level of 80 bits to present their results, and we did the same in chapter 5. It is however important to keep in mind that security can (and probably should) be increased for use in production, at the cost of an additional sacrifice in performance.

We can also encounter library-specific problems. For instance, the bit size of the moduli in the chain is not configurable in PALISADE. Because of this, the approximation error is sometimes too high at the last level, where the decryption is supposed to happen. In addition, if the scale is not high enough at intermediate levels, the error can also become too big.

Another issue is that HE libraries are not yet interoperable, which means that clients and servers must use the same library when exchanging keys and ciphertexts. Hopefully, this will be resolved in the future with the current effort on standardization.

Finally, it is also more challenging to set the numerous ML parameters. Without knowledge of the training data, a model owner has little means to evaluate a model and tune the parameters. They cannot observe the evolution of the loss at the end of an iteration, which makes it impossible to apply a convergence criterion. Solutions to these problems include having domain knowledge or using auxiliary datasets to find good parameter values.

Chapter 5

Evaluation

Now that the whole system has been presented in details in the two previous chapters, it is important to evaluate its performance and compare it to existing baselines. It will serve to determine if such a system is usable in practice. In this chapter, we provide benchmarks and analyses of the results for LR training and CNN inference with two different architectures.

5.1 Logistic regression

The following evaluation has first been carried out on a small dataset of 80 training samples and 20 test samples. The samples have 2 features and can be linearly separated in two classes. We also compare these results with a bigger dataset of 16000 training points and 4000 test points.

For the cryptographic parameters, we take a ring dimension of 2^{14}, a scale of 25 and a security level of 80 bits. The number of gradient descent iterations is 3, the learning rate at iteration t is the function $f(t) = 10/(t+1)$, and the degree of the polynomial approximation of the sigmoid is 3. These tests were run on an *AMD Ryzen 5 5600x* with 12 threads and 32 GB of RAM.

The total runtime and memory usage of the different pipelines are summarized in Table 5.1 and Figure 5.1. As a reminder, the *plaintext* pipeline uses the same algorithm as the two encrypted pipelines (*palisade* and *seal*), but without encryption. Two other pipelines serve as true baselines:

- *standard*: a simple implementation of a textbook logistic regression, without encryption.
- *sklearn*: a commonly used ML library.

Note that the small training dataset can be put entirely in 1 ciphertext, while it takes 10 ciphertexts to encrypt the big dataset.

Table 5.1: Total runtime of the LR pipelines

	plaintext	palisade	seal	standard	sklearn
Total runtime [s] - small	0.0028	1.5731	9.3481	0.0003	0.0056
Total runtime [s] - big	0.0104	6.2176	19.0390	0.0020	0.0093

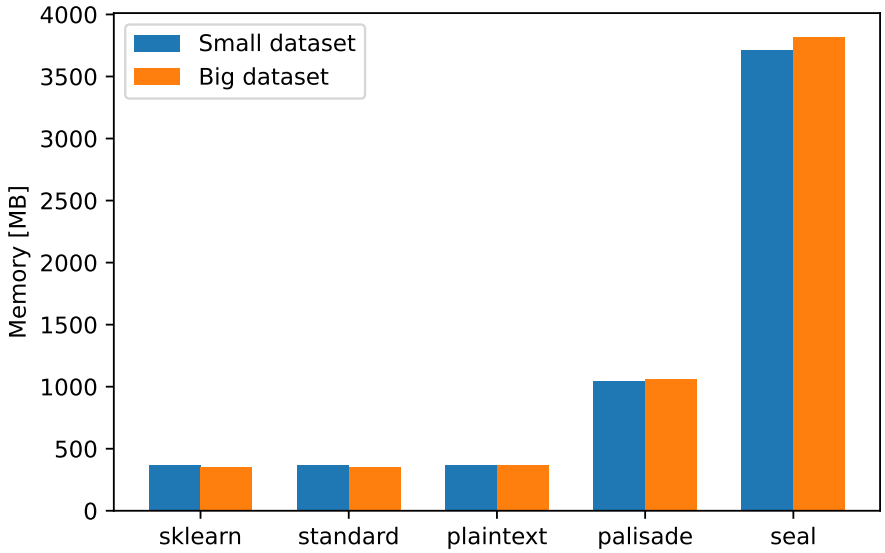

Figure 5.1: Memory footprint of the LR pipelines

We first observe that the encrypted pipelines are 3 to 4 orders of magnitude slower than the baselines. By comparing the plaintext and the standard pipelines, it appears that transforming the algorithm to a homomorphically-friendly version has a negligible performance cost relatively to the overhead of encryption. There is also not a big time difference between training on 80 points and on 16000 points, because part of the algorithm is independent on the number of input ciphertexts.

The model performance metrics (*accuracy* and *area under the curve*) are not different from the baseline (equal to 1 for the small dataset and close to 1 for the big one).

The approximate memory usage is almost identical for the two datasets. This indicates that it is dominated by the crypto context and not the ciphertexts. In particular, the rotation key can take a few gigabytes if the ring dimension is high. It also seems that SEAL consumes more memory than PALISADE, by a factor of at least 3 in this case.

Table 5.2 breaks down the runtime at every step (initialization, encryption, training, prediction, and decryption) of the different backends for the small dataset. PALISADE is faster than

SEAL by around a factor 2 for training and a factor 10 for the initialization. These are the two most time-consuming steps in this setting. Note that the encryption steps also include the packing of the data into vectors, which is why this step still takes a bit of time in the plaintext pipeline.

Table 5.2: Runtime of the LR at each step, in seconds

	Init	Encrypt train	Train	Encrypt test	Predict	Decrypt
plaintext	0.0001	0.0003	0.0019	0.0001	0.0001	0.0000
palisade	0.7835	0.0207	0.7309	0.0181	0.0146	0.0052
seal	7.6654	0.0311	1.6079	0.0316	0.0115	0.0006

5.2 Neural networks

Table 5.3: Architecture of an homomorphically encrypted network as a proof-of-concept

Layer	In chan.	Out chan.	Kernel	In neurons	Out neurons	Padding	Stride
Conv2d	1	3	5	28×28	24×24	0	0
AvgPool2d	3	3	3	24×24	8×8	0	3
Square	3	3		8×8	8×8		
Flatten							
Linear				192	10		

Let us now evaluate the performance of neural networks inference. The network architecture used in this section (Table 5.3) is not designed for maximal accuracy, rather it shows the differences between different layer types.

Table 5.4: Runtime of the CNN at each step, in seconds (pw = pixelwise and cw = channelwise)

	Init client	Encrypt	Init server	Predict	Decrypt	Serialize
plaintext-cw	0.0002	0.0002	0.0007	0.5351	0.0001	0.0003
plaintext-pw	0.0002	0.0250	0.0007	0.5090	0.0001	0.0287
seal-cw	0.5990	0.0062	0.0233	148.5469	0.0003	2.0905
seal-pw	0.2610	3.1043	0.0123	80.7692	0.0020	4.1557
pytorch			0.0026	0.0088		0.0000

The runtime of an encrypted inference with a convolutional neural network is presented in Table 5.4. For this setting, we omit the PALISADE pipeline and show only the SEAL and plaintext pipelines with both packing methods, along with a baseline using the `pytorch` library.

The other difference with section 5.1 is that we simulate a true client-server communication

and report the time needed to serialize and deserialize the request and the response. These steps are summed in one column *serialize* for readability. As introduced in section 3.3, the client needs to send the encrypted data, the relinearization key, and the rotation keys if needed. The server responds with the encrypted result. For this evaluation, these objects are encoded in `base64` and sent in JSON via HTTP.

We observe that the serialization step takes a considerable amount of time because the data and the keys take a lot of memory. However, the runtime is still dominated by the prediction step, which takes around 2 minutes for this simple network. The channelwise packing method was designed to reduce the latency for non-batched data, but it completely fails in this aspect by being even slower than pixelwise packing, except for the encryption step. This result is due to the high number of rotation operations and the higher multiplicative depth.

Table 5.5: Runtime of the CNN prediction at each layer, in seconds

	Convolution	Average pooling	Square	Linear
plaintext-channelwise	0.4752	0.0475	0.0002	0.0079
plaintext-pixelwise	0.4772	0.0077	0.0011	0.0184
seal-channelwise	138.6519	9.3982	0.0110	0.4806
seal-pixelwise	77.6176	0.5223	0.5011	2.1229

Table 5.5 breaks down the prediction step to show the impact of each layer type. However, it is important to keep in mind that homomorphic computations are slower on the first levels, with fresh ciphertexts. Thus, a layer placed at the beginning of a network will be slower than its equivalent at the end of the network. This is although not the case for the plaintext pipeline, which indicates that the convolutional layer is still the slowest by far, at least in the plaintext domain. It also shows that channelwise packing has an advantage for linear layers and activation functions, compared to pixelwise packing.

In terms of memory usage, shown in Figure 5.2, channelwise packing has a clear advantage. It indicates that channelwise packing, even if it is slower, can be useful for systems with limited memory. However, it is important to note that both the memory usage and the runtime will grow linearly with channelwise packing, while pixelwise packing will be able to batch multiple data points at no additional cost.

5.3 Hybrid CNN

In this section, we use a standard network architecture to see how our framework can be applied to a real use-case. These tests were run on an Azure `D16as_v4` machine with 16 threads and 64 GB of RAM.

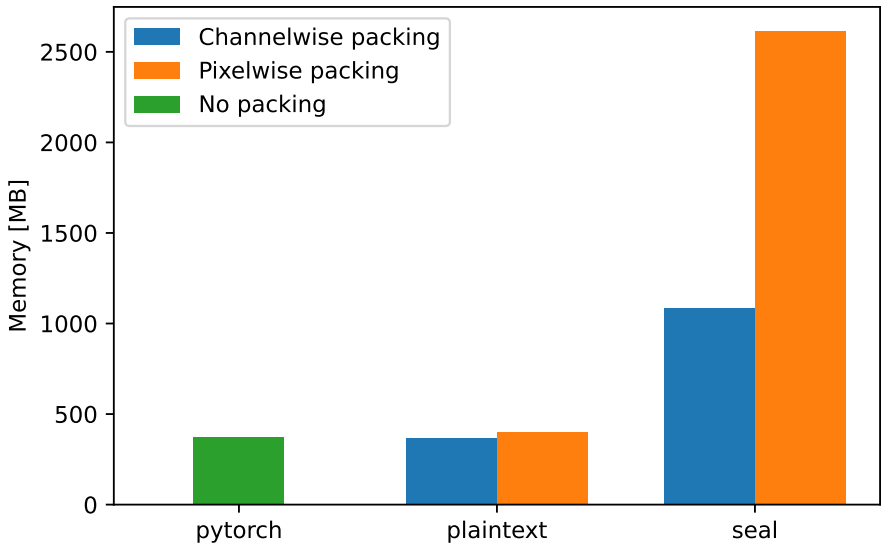

Figure 5.2: Memory footprint of the CNN pipelines

It should be accurate enough while still having reasonable performance. Judging from the results of section 5.2, we need to choose a very simple architecture. The network is identical to LeNet-5 by LeCun [31], except that we replace the $\tanh$ layers with a polynomial approximation of degree 3. Layers are described in Table 5.6.

Table 5.6: Architecture of a homomorphically encrypted LeNet-5

Layer	In chan.	Out chan.	Kernel	In neurons	Out neurons	Padding	Stride
Conv2d	1	6	5	28×28	28×28	2	0
TanhApprox							
AvgPool2d	6	6	2	28×28	14×14	0	2
Conv2d	6	16	5	14×14	10×10	0	0
TanhApprox							
AvgPool2d	16	16	2	10×10	5×5	0	2
Flatten							
Linear				400	120		
TanhApprox							
Linear				120	84		
TanhApprox							
Linear				84	10		

We also consider a hybrid version, as described in subsection 3.3.3, where only the last five layers after flattening are in the encrypted domain. In this case, the two first $\tanh$ layers do not need to be approximated and the learning rate is adapted accordingly.

The model is trained on the MNIST dataset, and the test samples are batched together in the same ciphertext with pixelwise packing. This lets us compute and compare the accuracy of the different versions in Table 5.7, along with the runtime and memory usage.

The results indicate that even a simple network needs a lot of resources, with a latency of 176 minutes and over 50 GB of RAM used (in the case of SEAL). However, if this latency can be tolerated, the time per inference is only of about 1.3 seconds with a batch of 8192 samples.

Furthermore, the hybrid version has a much smaller footprint and is also much faster, with an amortized time per inference of only 1 millisecond. When it comes to accuracy, we notice a slight decrease due to the imprecisions in the encrypted computations.

Table 5.7: Results for LeNet-5

	Runtime [min]	Memory [MB]	Accuracy
pytorch	0.005	392	0.976
plaintext-pixelwise	0.136	379	0.976
hybrid-seal-pixelwise	8.570	2210	0.964
seal-pixelwise	176.000	51800	0.953

Chapter 6

Related work

While some of the design and implementation decisions presented in the previous chapters are new, many other papers already suggest solutions for homomorphically encrypted machine learning.

One of the most influential is CryptoNets [20], which is a 5-layers CNN evaluated on the MNIST dataset. The packing method used is what we call *pixelwise packing* in this work. They also replace max-pooling layers with scaled mean-pooling layers, and ReLU with a square activation function. They choose YASHE as the encryption scheme, which is not commonly used today. It inspired many other works that attempted to include optimizations to alleviate the performance and memory limitations.

For instance, Badawi et al. [32] implement an optimization using GPUs and evaluate their results on both the MNIST and CIFAR-10 datasets. In [33], they design an algorithm for a specific network architecture of 2 fully connected layers (*fasttext* for text classification). We started with this architecture before moving to a more generic framework.

Similarly to our hybrid model, LoLa [21] first computes a low dimension representation before encrypting it. It then selects an optimal packing method at each layer. It also chooses an optimal matrix-vector multiplication algorithm based on the packing. In our framework, the packing type does not change across the layers, because it would be difficult to select automatically.

Lee et al. [34] focus on a maximal accuracy by using a deep model (ResNet-20) on CIFAR-10. It is one of the few works that uses bootstrapping.

Some works tackle NN training, but this requires either to use bootstrapping, or to delegate the weights update to the client [35] or to a semi-honest server [36].

Surveys [37, 38] of privacy-preserving neural networks summarize these works, list the available libraries and explain the challenges. Compared to all these solutions, our design is meant to

be flexible to allow finding the desired trade-off between performance and utility.

Regarding logistic regression training, Kim et al. [19] heavily inspired this work, with the exception of a few generalizations on the input size and the use of regularization. Another difference is that they used the HEAAN library. Other solutions include Bonte and Vercauteren [39] and Chen et al. [40]. Both are based on the BFV encryption scheme.

Although NN and LR are the most studied models, there are also solutions for decision trees [41] and SVMs [42]. It is always good to have a variety of models at hand, because each has its own advantages and disadvantages.

Chapter 7

Conclusion

Starting from a wide range of threats related to ML systems, we have identified two important threats that can be mitigated using homomorphic encryption. The suggested solutions take into account the practical constraints of a real-world deployement, such as runtime, memory usage and accuracy. We evaluated different alternatives that can be used depending on the context.

From the results, we have seen that even simple models like logistic regression are several orders of magnitude slower to train, and require a lot of memory compared to a cleartext version. The same applies for neural networks. Even by protecting only the inference phase, it is still prohibitively expensive for many use-cases involving deep networks and big images.

We then conclude that homomorphic encryption is still an emerging technology from the academia, and it still needs improvements before being usable in a real-world application. In particular, machine learning is a resource-intensive process that suffers greatly from the overhead of encryption.

However, the potential threats that would be solved by these solutions are sufficiently important to justify continuing the efforts in this field. Optimizations can be found either at the cryptographic level or at the algorithmic level of ML models, for instance by using different packing methods or approximations. Parallelization can also be extremely valuable, and the desired security level has a high impact on performance as well.

Another promising paradigm is to delegate some computations to the client and to keep the encrypted parts of the pipeline to a strict minimum while still providing enough security. The appropriate trade-off needs to be found on a case-by-case basis, depending on the capabilities of the client and those of the adversary.

It is also worth noting that future work could be done to extend our system to a multi-party setting by using MHE or MKHE. However, other approaches for multi-party training are also being researched. Instead of collecting all the data to a central server, *Federated Learning* consists

of training multiple local models by each data owner, and combining these models into one. This way, the data never leaves its owner, even encrypted. As only model weights are exchanged, this could remove the need of encryption entirely, but weights can still retain sensitive information. Thus, some works such as POSEIDON [43] combine this idea with MHE for better security guarantees. Once again, the ideal approach highly depends on the use-case.

Bibliography

[1] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. "ON DATA BANKS AND PRIVACY HOMOMORPHISMS". In: *Foundations of secure computation* (1978).

[2] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Vol. 1592. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-65889-4. DOI: 10.1007/3-540-48910-X_16.

[3] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*. the 41st annual ACM symposium. Bethesda, MD, USA: ACM Press, 2009, p. 169. ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536440.

[4] Zvika Brakerski, Craig Gentry, and V. Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ITCS '12*. 2012. DOI: 10.1145/2090236.2090262.

[5] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption". In: *Cryptology ePrint Archive* (2012).

[6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Cham: Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70693-1 978-3-319-70694-8. DOI: 10.1007/978-3-319-70694-8_15.

[7] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "A Full RNS Variant of Approximate Homomorphic Encryption". In: *Selected Areas in Cryptography – SAC 2018*. Ed. by Carlos Cid and Michael J. Jacobson. Vol. 11349. Cham: Springer International Publishing, 2019, pp. 347–368. ISBN: 978-3-030-10969-1 978-3-030-10970-7. DOI: 10.1007/978-3-030-10970-7_16.

[8] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "Bootstrapping for Approximate Homomorphic Encryption". In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. Cham: Springer International Publishing, 2018, pp. 360–384. ISBN: 978-3-319-78380-2 978-3-319-78381-9. DOI: 10.1007/978-3-319-78381-9_14.

[9] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. “Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys”. In: *Advances in Cryptology – EUROCRYPT 2021*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. Cham: Springer International Publishing, 2021, pp. 587–617. ISBN: 978-3-030-77869-9 978-3-030-77870-5. DOI: 10.1007/978-3-030-77870-5_21.

[10] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. “On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption”. In: *Proceedings of the 44th symposium on Theory of Computing - STOC ’12*. the 44th symposium. New York, New York, USA: ACM Press, 2012, p. 1219. ISBN: 978-1-4503-1245-5. DOI: 10.1145/2213977.2214086.

[11] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. “Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19: 2019 ACM SIGSAC Conference on Computer and Communications Security. London United Kingdom: ACM, Nov. 6, 2019, pp. 395–412. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363207.

[12] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. “Multiparty Homomorphic Encryption from Ring-Learning-with-Errors”. In: *Proceedings on Privacy Enhancing Technologies* 2021.4 (Oct. 1, 2021), pp. 291–311. ISSN: 2299-0984. DOI: 10.2478/popets-2021-0071.

[13] Qiang Liu, Pan Li, Wentao Zhao, Wei Cai, Shui Yu, and Victor C. M. Leung. “A Survey on Security Threats and Defensive Techniques of Machine Learning: A Data Driven View”. In: *IEEE Access* 6 (2018), pp. 12103–12117. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2805680.

[14] Maria Rigaki and Sebastian Garcia. *A Survey of Privacy Attacks in Machine Learning*. Apr. 1, 2021. DOI: 10.48550/arXiv.2007.07646. arXiv: 2007.07646[cs].

[15] Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A. Erdogdu, and Ross Anderson. *Manipulating SGD with Data Ordering Attacks*. June 5, 2021. DOI: 10.48550/arXiv.2104.09667. arXiv: 2104.09667[cs].

[16] Seira Hidano, Takao Murakami, Shuichi Katsumata, Shinsaku Kiyomoto, and Goichiro Hanaoka. “Model Inversion Attacks for Prediction Systems: Without Knowledge of Non-Sensitive Attributes”. In: *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. 2017 15th Annual Conference on Privacy, Security and Trust (PST). Calgary, AB: IEEE, Aug. 2017, pp. 115–11509. ISBN: 978-1-5386-2487-6. DOI: 10.1109/PST.2017.00023.

[17] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. “Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS’15: The 22nd ACM Conference on Computer and Communications Security. Denver Colorado USA:

ACM, Oct. 12, 2015, pp. 1322–1333. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813677.

[18] Kuan-Chieh Wang, Yan Fu, Ke Li, Ashish Khisti, Richard Zemel, and Alireza Makhzani. *Variational Model Inversion Attacks*. Jan. 26, 2022. DOI: 10.48550/arXiv.2201.10787. arXiv: 2201.10787[cs].

[19] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. "Logistic regression model training based on the approximate homomorphic encryption". In: *BMC Medical Genomics* 11.4 (Oct. 11, 2018), p. 83. ISSN: 1755-8794. DOI: 10.1186/s12920-018-0401-7.

[20] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. "CryptoNets: applying neural networks to encrypted data with high throughput and accuracy". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, June 19, 2016, pp. 201–210.

[21] Alon Brutzkus, Oren Elisha, and Ran Gilad-Bachrach. *Low Latency Privacy Preserving Inference*. June 6, 2019. DOI: 10.48550/arXiv.1812.10659. arXiv: 1812.10659[cs,stat].

[22] Microsoft Corporation. *Microsoft SEAL*. 2022. URL: https://github.com/microsoft/SEAL.

[23] PALISADE. *PALISADE*. 2022. URL: https://gitlab.com/palisade/palisade-release.

[24] CryptoLab inc. *HEAAN*. 2022. URL: https://github.com/snucrypto/HEAAN.

[25] Tune Insight SA and EPFL-LDS. *Lattigo v3*. 2022. URL: https://github.com/tuneinsight/lattigo.

[26] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "SoK: Fully Homomorphic Encryption Compilers". In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 1092–1108. DOI: 10.1109/SP40001.2021.00068. arXiv: 2101.07078[cs].

[27] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. "Homomorphic Encryption Standard". In: *Protecting Privacy through Homomorphic Encryption*. Ed. by Kristin Lauter, Wei Dai, and Kim Laine. Cham: Springer International Publishing, 2021, pp. 31–62. ISBN: 978-3-030-77286-4 978-3-030-77287-1. DOI: 10.1007/978-3-030-77287-1_2.

[28] HuGang. *SEAL-Python*. 2022. URL: https://github.com/Huelse/SEAL-Python.

[29] Robert Podschwadt and Daniel Takabi. "Classification of Encrypted Word Embeddings using Recurrent Neural Networks". In: *Proceedings of the PrivateNLP 2020: Workshop on Privacy in Natural Language Processing*. 2020.

[30] Diego Chialva and Ann Dooms. *Conditionals in Homomorphic Encryption and Machine Learning Applications*. May 9, 2019. arXiv: 1810.12380[cs].

[31] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86 (Dec. 1, 1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[32] Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Jun Jie Sim, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. *Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs*. Aug. 18, 2020. DOI: 10.48550/arXiv.1811.00778. arXiv: 1811.00778[cs].

[33] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. "PrivFT: Private and Fast Text Classification With Homomorphic Encryption". In: *IEEE Access* 8 (2020), pp. 226544–226556. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3045465.

[34] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. *Privacy-Preserving Machine Learning with Fully Homomorphic Encryption for Deep Neural Network*. June 14, 2021. DOI: 10.48550/arXiv.2106.07229. arXiv: 2106.07229[cs].

[35] Kentaro Mihara, Ryohei Yamaguchi, Miguel Mitsuishi, and Yusuke Maruyama. *Neural Network Training With Homomorphic Encryption*. Dec. 25, 2020. DOI: 10.48550/arXiv.2012.13552. arXiv: 2012.13552[cs].

[36] Reda Bellafqira, Gouenou Coatrieux, Emmanuelle Genin, and Michel Cozic. *Secure Multilayer Perceptron Based On Homomorphic Encryption*. June 7, 2018. DOI: 10.48550/arXiv.1806.02709. arXiv: 1806.02709[cs].

[37] Bernardo Pulido-Gaytan, Andrei Tchernykh, Jorge M. Cortés-Mendoza, Mikhail Babenko, Gleb Radchenko, Arutyun Avetisyan, and Alexander Yu Drozdov. "Privacy-preserving neural networks with Homomorphic encryption: Challenges and opportunities". In: *Peer-to-Peer Networking and Applications* 14.3 (May 1, 2021), pp. 1666–1691. ISSN: 1936-6450. DOI: 10.1007/s12083-021-01076-8.

[38] Robert Podschwadt, Daniel Takabi, and Peizhao Hu. *SoK: Privacy-preserving Deep Learning with Homomorphic Encryption*. Jan. 1, 2022. DOI: 10.48550/arXiv.2112.12855. arXiv: 2112.12855[cs].

[39] Charlotte Bonte and Frederik Vercauteren. "Privacy-preserving logistic regression training". In: *BMC Medical Genomics* 11 (S4 Oct. 2018), p. 86. ISSN: 1755-8794. DOI: 10.1186/s12920-018-0398-y.

[40] Hao Chen, Ran Gilad-Bachrach, Kyoohyung Han, Zhicong Huang, Amir Jalali, Kim Laine, and Kristin Lauter. "Logistic regression over encrypted data from fully homomorphic encryption". In: *BMC Medical Genomics* 11 (S4 Oct. 2018), p. 81. ISSN: 1755-8794. DOI: 10.1186/s12920-018-0397-z.

[41] Adi Akavia, Max Leibovich, Yehezkel S. Resheff, Roey Ron, Moni Shahar, and Margarita Vald. *Privacy-Preserving Decision Tree Training and Prediction against Malicious Server*. 1282. 2019.

[42] Saerom Park, Junyoung Byun, Joohee Lee, Jung Hee Cheon, and Jaewook Lee. “HE-Friendly Algorithm for Privacy-Preserving SVM Training”. In: *IEEE Access* 8 (2020), pp. 57414–57425. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2981818.

[43] Sinem Sav, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. “POSEIDON: Privacy-Preserving Federated Neural Network Learning”. In: *Proceedings 2021 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. Virtual: Internet Society, 2021. ISBN: 978-1-891562-66-2. DOI: 10.14722/ndss.2021.24119.

Appendix A

Algorithms

Here we describe the main algorithms used in this project. It is meant to illustrate the implementation of an algorithm with homomorphic encryption and CKKS, rather than being an exhaustive reference of all the algorithms implemented. Note that homomorphic operations (**Mult**, **Add**, **Rotate**) are abstracted as mentioned in section 4.1, i.e. ciphertexts are automatically rescaled and relinearized after every multiplication when necessary.

In this type of algorithms, the computation of dot products is very common. It is done by first multiplying the two vectors, and successively adding the result with a rotated copy of the result. This outputs a vector containing the dot product at every element, but it can be masked if it needs to be only at one position.

By doing rotations of successive powers of two, the computational complexity of dot products can be reduced to $O(\log n)$ instead of $O(n)$. This is the reason why vectors are often padded with zeros to the next power of two.

A.1 Logistic regression training

Algorithm 1 describes how to train an encrypted LR model, from the encryption of the training data to the decryption of the trained weights. It is inspired by [19] but with less assumptions on the input size, and with the addition of a regularization term. For simplicity, we use traditional gradient descent instead of Nesterov's gradient update.

Algorithm 1: Logistic regression training

Input: A matrix of samples $\mathbf{X} \in \mathbb{R}^{n \times (d-1)}$, the labels $\mathbf{y} \in \mathbb{R}^n$, and the ring degree N
Output: The trained weights $\beta \in \mathbb{R}^d$
Precondition: $d \leq N/2$

Let s be the number of slots $N/2$
`// Encryption`
Add a leading column of ones to $\mathbf{X}$
Pairwise multiply the points of $\mathbf{X}$ with their corresponding label in $\mathbf{y}$, as a matrix $\mathbf{Z}$
Pad the columns of $\mathbf{Z}$ with zeros to the next power of two d_{pad}
Encrypt the matrix $\mathbf{Z}$ row by row into a set of ciphertexts $\{c_{z_i}\}_{i=1}^{L}$
Let $r = 2^{\lceil \log_2 \min(n, s/d_{pad}) \rceil}$ be the number of rows in each ciphertext
`// Training`
Initialize a vector of size s with random Xavier weights duplicated at each interval d_{pad}, and encrypt it into c_β
Let $\mathbf{SigmoidApprox}$ be a sigmoid approximation function as defined in [19]
Let $x = [x_1, ..., x_s]$ be a mask vector keeping only the first element of each row, where $x_u = 1$ if $(u-1) \mod d_{pad} = 0$, and $x_u = 0$ otherwise
Let p a regularization vector with each non-padded element set to $-2/C$
for $t = 1, ..., T$ **do**
    for $i = 1, ..., L$ **do**
        $c_i \leftarrow \mathbf{Mult}(c_{z_i}, c_\beta)$
        for $j = 0, ..., \log_2(d_{pad}) - 1$ **do**
            $c_i \leftarrow \mathbf{Add}(c_i, \mathbf{Rotate}(c_i, 2^j))$
        end
        $c_i \leftarrow \mathbf{Mult}(c_i, x)$
        for $j = 0, ..., \log_2(d_{pad}) - 1$ **do**
            $c_i \leftarrow \mathbf{Add}(c_i, \mathbf{Rotate}(c_i, -2^j))$
        end
        $c_i \leftarrow \mathbf{SigmoidApprox}(c_i)$
        $c_i \leftarrow \mathbf{Mult}(c_i, c_{z_i})$
    end
    $c \leftarrow \mathbf{Sum}(\{c_i\}_{i=1}^{L})$
    for $j = \log_2(d_{pad}), ..., \log_2(d_{pad}) + \log_2(r) - 1$ **do**
        $c \leftarrow \mathbf{Add}(c, \mathbf{Rotate}(c, 2^j))$
    end
    $c \leftarrow \mathbf{Add}(c, \mathbf{Mult}(c_\beta, p))$
    Let l a learning rate vector with each non-padded element set to the learning rate at iteration t
    $c_\beta \leftarrow \mathbf{Add}(c_\beta, \mathbf{Mult}(c, l))$
end
`// Decryption`
Decrypt c_β into β and truncate the vector to the first d elements

A.2 2D convolution layer

Algorithm 2 shows how to evaluate an encrypted 2D convolution layer that is part of a neural network. It uses *channelwise* packing.

For more readability, this version does not support padding. The output feature maps are thus smaller than the input.

The version for *pixelwise* packing is much simpler, as it only requires one homomorphic multiplication and addition in the inner loop. It is thus omitted here.

Algorithm 2: 2D convolution layer (channelwise packing)

Input: A collection of ciphertexts for B samples and W_{in} input channels where each ciphertext represents a row-wise encoded image of size $R_{in} \times C_{in}$, a collection of kernels and biases, and the ring degree N
Output: A collection d of ciphertexts for B samples and W_{out} output channels
Precondition: $R_{in} \cdot C_{in} \leq N/2$

for $b = 1, ..., B$ **do**
  for $m = 1, ..., W_{out}$ **do**
    for $l = 1, ..., W_{in}$ **do**
      Let k be the kernel for input channel w_{in} and output channel w_{out}, padded with zeros to a shape of $R_{in} \times C_{in}$ and row-wise encoded
      Let x be the ciphertext for sample b and input channel w_{in}
      for $i = 1, ..., R_{out}$ **do**
        for $j = 1, ..., C_{out}$ **do**
          Let k_{rot} be the kernel k rotated by $i \cdot C_{in} + j$ positions to the right
          $z \leftarrow \mathbf{Mult}(x, k_{rot})$
          for $q = 0, ..., \log_2(N/2) - 1$ **do**
            $z \leftarrow \mathbf{Add}(z, \mathbf{Rotate}(z, -2^q))$
          end
          Let a be a zero vector with element $i \cdot C_{out} + j$ set to 1
          $z \leftarrow \mathbf{Mult}(z, a)$
          $d_{bm} \leftarrow \mathbf{Add}(d_{bm}, z)$
        end
      end
    end
    Let y be a vector filled with the bias value for channel m
    $d_{bm} \leftarrow \mathbf{Add}(d_{bm}, y)$
  end
end