FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme / specialisation: Data Science	The spring semester, 2023 Open / ~~Confidential~~
Author: Emil Alan Pierzgalski	
Supervisor at UiS: Ferhat Özgur Catak	
Thesis title: Privacy of 5G Enabled Networks: Homomorphic Encryption based Privacy-Preserving Machine Learning	
Credits (ECTS): 30	
Keywords: Spectrum Sensing, Machine Learning, Federated Learning, Homomorphic Encryption	Pages: 31 + appendix: Link to repository Stavanger, July 15, 2023

Approved by the Dean 30 Sep 21
Faculty of Science and Technology

Abstract

Homomorphic encryption (HE) is a technique that allows computations to be performed on encrypted data, just as if the data were unencrypted. This has numerous potential applications, such as sensitive medical data, mainly when privacy and anonymity are critical. HE can also be used in cases where multiple parties need to perform computations on shared data without revealing the data to one another. One fascinating application of HE is in machine learning, specifically in a process known as federated learning (FL). FL is a cutting-edge method that is particularly useful in situations where privacy is essential, as it eliminates the need for data to be shared with a central server, as is the case with traditional distributed machine learning models. However, privacy risks are associated with sharing model parameters, as inference attacks can obtain sensitive information. This issue can be addressed by encrypting the model parameters with HE on the client side and aggregating the encrypted data. In this paper, we explore federated learning with homomorphic encryption to improve the privacy of 5G networks. The results of our experiments show that encryption has a minimal effect on the the predictive performance of the model, but leads to an increase in computation time by 587 %, 624 % and 679 % for 2, 5, and 7 clients, respectively.

Acknowledgements

I would like to express my gratitude to my supervisor, Ferhat Özgur Catak, for introducing me to the main topics of the project, as well as his continuous guidance and excellent feedback along the way.

Contents

Acronyms

CKKS Cheong-Kim-Kim-Song.

CNN Convolutional Neural Network.

DL Deep Learning.

FL Federated Learning.

HE Homomorphic Encryption.

IoT Internet of Things.

IoU Intersection over Union.

ML Machine Learning.

Chapter 1

Introduction

1.1 Background

Over the past decades, there have been rapid advancements in networking technology. This progression has allowed many devices to be connected. However, as the number of devices grow, the wireless spectrum becomes more occupied due to its finite nature. Therefore, the wireless spectrum needs to be utilized effectively. The problem of efficiently utilizing the wireless spectrum, known spectrum sensing, is a major challenge in cognitive radios (CRs) [1].

In recent years, deep learning (DL), has been demonstrated as a method for spectrum sensing [2, 3]. Using a convolutional neural network (CNN), the authors in [2] demonstrate that their model can achieve the state of art detection performance, requiring no prior knowledge about the channel state information or background noise, unlike traditional techniques.

A major challenge of utilizing DL models is the need of a large amount of training data to make reliable predictions. Typically, these models are trained in the cloud, using a large amount of user data. However, this entails potential privacy and security risks, as the data needs to be shared with a third party.

To address this problem, the researchers at Google proposed a new method called federated learning (FL) in 2017 [4]. This method allows for training a decentralized model without needing to share the user data. The data can remain on edge devices, with needing to only share the model parameters.

Despite being a major improvement in terms of privacy, sending model parameters can also entail privacy risks. Previous research has shown that adversaries can expose sensitive user information by doing inference attacks on the models parameters [5].

One way of protecting the parameters is by using homomorphic encryption (HE) [6, 7]. With this approach, the weights are encrypted ahead of time locally, and only the the encrypted parameters are sent to the server, adding an extra layer of security to federated learning.

1.2 Problem Statement

Deep learning is a potential solution to making spectrum sensing more effective than the previous methods. However, the large amount of data needed to make such models reliable comes with a privacy risk. This project aims to address this challenge by utilizing federated learning and a variant of homomorphic encryption, CKKS [8].

1.3 Outline and Contribution

The contribution in this project are threefold. Firstly, we do a comparison of various PyTorch models on a spectrum dataset. Furthermore, we pick a best suited model and measure its performance in a FL setting. Lastly, we add homomorphic encryption, and measure its impact on performance using various encryption parameters.

- Chapter 2 provides a theoretical overview of the key concepts used in the report.
- Chapter 3 describes the experiment setup and mentions the required tools for experiments.
- Chapter 4 presents the results of the experiments from chapter 3.
- Chapter 5 summarizes the results.
- Chapter 6 concludes the report.

Chapter 2

Theory

2.1 Homomorphic Encryption

Homomorphic encryption (HE) is a type of encryption that enables computations on encrypted data, yielding the same value if the computation was done on unencrypted data. A comprehensive overview of the subject can be found in the survey [9]. This following summary of HE based on that source:

The idea of homomorphic encryption itself is not a new concept. It dates back to 1978 when it was first introduced by Rivest et al. [10]. However the approach was limited by only enabling either one type or a limited amount of computations on the encrypted data.

In 2009, a breakthrough was made by Gentry [11], creating a new approach to HE, enabling both computations for an unlimited amount of times. This is also known as fully homomorphic encryption (FHE).

In general, we distinguish between three types of HE: Partially Homomorphic Encryption (PHE) allows for only one operation: either addition or multiplication, but not both. Somewhat Homomorphic Encryption (SHE) allows for both addition and multiplication, but only for a limited amount of times. Fully Homomorphic Encryption (FHE) allows for both operations for a unlimited number of times.

Although Gentry's method to FHE was groundbreaking, his method uses a

process known as bootstrapping which makes the scheme computationally expensive and often not feasible in real world scenarios. As a result numerous different schemes have emerged in the following years. In particular, a fairly recent scheme, CKKS, will be a part of this project.

2.1.1 CKKS Scheme

The Cheon-Kim-Kim-Song (CKKS) scheme is a form of HE that enables arithmetic operations such as addition or multiplications on real numbers [8]. While the details of CKKS scheme can be complex and require an understanding of advanced mathematics and cryptography, a quite comprehensive summary can be found in [12]. This paper summarizes all of main components of the CKKS scheme including *encoding and decoding*, *key generation*, *encryption and decryption*, *evaluation*, and arguably its most important feature, *rescaling*. The following formulas are obtained from that source.

Encoding and decoding. To encrypt a message with CKKS, it has to be encoded first. Encoding works by mapping a vector of complex numbers, z to a plaintext object, a. For this, a scale factor Δ and a canonical embedding function π are used. The encoding function is given by:

$$Encode(z, \Delta) = [\Delta \cdot \pi^{-1}(z)] \tag{2.1}$$

The encoded result will be an element of the polynomial ring $R_q = \mathbf{Z}_q[x]/f(x)$, where q is a prime number used for modulus and $f(x)$ is a polynomial.

The decoding is just the opposite of encoding. It maps the plaintext objects to a vector of complex numbers. The decoding function can be written as:

$$Decode(a, \Delta) = \pi(\frac{1}{\Delta} \cdot a) \tag{2.2}$$

Key generation. The key generation is done accordingly. A secret key Sk is sampled from a polynomial of degree n with coefficients {-1, 0, +1}. Then, the public key are calculated as follows:

$$\begin{aligned} Pk_1 &= [-a \cdot Sk + e]_{qL} \\ Pk_2 &= a \xleftarrow{U} R_{qL} \end{aligned} \tag{2.3}$$

where a is a random polynomial sampled uniformly from R_{qL}, and e is an error term. The $[\cdot]_{qL}$ denotes a modulo operation by prime q.

Encryption and decryption. After the keys have been generated, encryption can be performed. To encrypt a plaintext message M, the following formula is used:

$$\begin{aligned} C_1 &= [Pk_1 \cdot u + e_1 + M]_{qL} \\ C_2 &= [Pk_2 \cdot u + e_2]_{qL} \end{aligned} \tag{2.4}$$

where u are random polynomials from R_2 and e_1 and e_2 are error terms. The results are then combined to form a ciphertext $C = (C_1, C_2)$ in R_{ql}^2

To decrypt a ciphertext, a secret key is needed. The result will give an approximate value of M:

$$\hat{M} = [C_1 + C_2 \cdot Sk]_{ql} \tag{2.5}$$

Evaluation. The CKKS scheme offers two evaluation functions on ciphertexts: multiplication and addition. The function for addition is given as:

$$\begin{aligned} EvalAdd(C^{(1)}, C^{(2)}) =& ([C_1^{(1)} + C_1^{(2)}]_{ql}, [C_2^{(1)} + C_2^{(2)}]_{ql}) \\ &= (C_1^{(3)}, C_2^{(3)}) = C^{(3)} \end{aligned} \tag{2.6}$$

while the function for multiplication is given as:

$$\begin{aligned} EvalMult(C^{(1)}, C^{(2)}) =& ([C_1^{(1)} \cdot C_1^{(2)}]_{ql}, [C_1^{(1)} \cdot C_2^{(2)} + C_2^{(1)} \cdot C_1^{(2)}]_{ql}, \\ & [C_2^{(1)} \cdot C_2^{(2)}]_{ql}) = (C_1^{(3)}, C_2^{(3)}, C_3^{(3)}) = C^{(3)} \end{aligned} \tag{2.7}$$

Rescaling is an important part of the CKKS scheme, and is what contributes to reducing the noise levels. As seen from the evaluation function, the multiplication causes the polynomial size to grow by one, and thus also leading to a larger scale. If the scale factor grows too big, it can have an impact on evaluation and provide wrong results. To keep the scale factor down, rescaling needs to be performed:

$$Rescale(C, \Delta) = \frac{1}{\Delta} \cdot [C_1, C_2]_{ql} = [\hat{C}]_{ql-1} \tag{2.8}$$

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of neural networks that are used to process image data. Generally, they can be summarized as convolutional layers, pooling layers and fully connected layers [13]:

- Convolutional layers are the fundamental features of CNNs. These layers are responsible for extracting the features of the data. For that purpose they utilize learnable kernels, which are small matricies. Once the layer receives an input, the kernel is mapped onto it, where it is then used 'slide' to over the whole spatial dimension. For every step in the sliding process, a dot product is calculated between the input and the kernel resulting in activation maps. These activations are then followed up with activation functions. A common one is rectified linear unit, better known as ReLu.

  To reduce complexity of the model, a few parameters of the convolutional layer can be set. These are depth, step-size and zero-padding.

- Pooling layers are aimed at reducing dimensionality and thus the computational complexity of the model. They function by covering each activation map and applying an operation on it. A common operation is max function, better known by max-pooling. There are also other other pooling operations such as average-pooling.

- Fully connected layers are layers that have every neuron of one layer connected to every neuron of the next layers, hence their name.

In essence, a CNN takes an input vector and passes it through a sequence of layers. These layers are responsible for extracting and learning features from the data, producing a final result. The nature of the result varies on the specific task of the CNN. For instance, in object classification tasks, the CNN given image, is used to assign a label to an image indicating its content, e.g. cat or a dog [14].

On the other hand, another application is called *semantic segmentation*. The aim of semantic segmentation, unlike object detection, is not to label an entire object, but rather the individual pixels [15]. In this project we will utilize semantic segmentation to classify pixels in wireless spectrograms into one of three possible classes: LTE signals, 5G NR signals, or noise signals.

2.3 Federated Learning

Federated Learning (FL) is a type of distributed machine learning that enables entities to train a global model without sharing their local data. Since the origin of FL, there have been proposed many variants of FL. The most common one, however, is called FederatedAveraging or FedAvg [4].

FedAvg can be summarized as follows. The participants train a model for a certain amount of rounds, producing a locally updated model. Then, the models parameters are sent to a central server where they are averaged to produce a global model. This averaging is weighted to ensure that models with a large amount training data will have a greater influence over the models parameters. After the averaging step, the global model is sent back to the clients. This concludes one round of FL. The process is done iteratively for an arbitrary amount of rounds, usually until the model converges.

FL has these key properties that make it an attractive option compared to other methods [4]:

- It can handle a large amount clients simultaneously. The authors expect the number of participating clients be much larger than the average number of training samples per client.
- It is designed to handle non-IID data. Because clients hold a different set of data, the distribution of data might vary among them. For instance, if training a language model, the difference in distribution may be influenced by each clients unique vocabulary.
- The size of data may be different from one client to another. FL is designed to handle these imbalances.
- FL aims to limit the communication between clients, as the amount of speed and availability might vary across the clients. It does so by sending the parameters which are a lot smaller in size than raw data.

The nature of FL, making the raw data stay on the local device, makes FL a much more privacy friendly option than traditional methods. However, one downside is that if one server is compromised and an unwanted party gets access to to the parameters, it can lead to inference attacks [5].

In this project we will be adding an additional layer of security by encrypting the weights before sending them with HE.

2.4 Spectrum Sensing

Spectrum sensing is a method for detecting the presence of primary users within a licensed spectrum, and is a key aspect of of cognitive radios. When secondary users want to utilize a licensed band, they must perform spectrum sensing to detect the presence of primary users. If the presence is detected, the secondary users have to change channel or reduce transmit power to not cause interference [16].

Chapter 3

Methodology

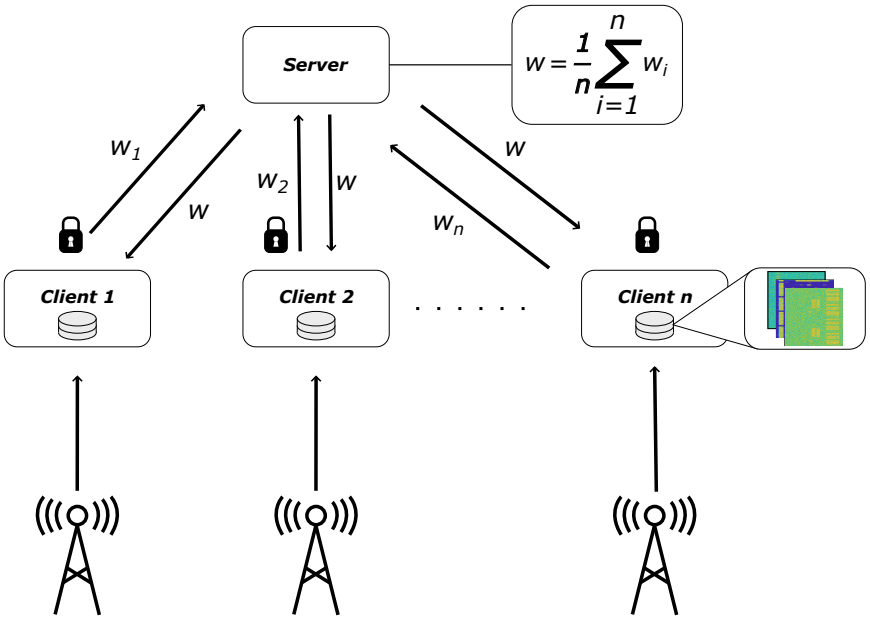

Figure 3.1: An overview of the proposed model.

In this chapter, we explain our approach. Figure 3.1, provides a simplified overview of how it might be applied to a real-world setting. Each client collect

wireless spectrum data from a nearby cell tower to train a local model. Next, the weights are encrypted and transmitted to a central server for aggregation, using the FedAvg approach. Upon receiving the averaged weights from the server, they can be combined with the local model. The result is an improved model as every client indirectly benefits from the data of others, while preserving privacy by avoiding the sharing of raw data.

3.1 Data

The dataset is downloaded by following the Matlab tutorial on deep learning for spectrum monitoring in [17]. In total, it consists of 900 256 x 256 images of wireless signals and their corresponding ground truth labels, where each image consists of three types of signals: 5G NR, LTE and noise signals. A sample of three wireless frames is shown in the figure 3.2.

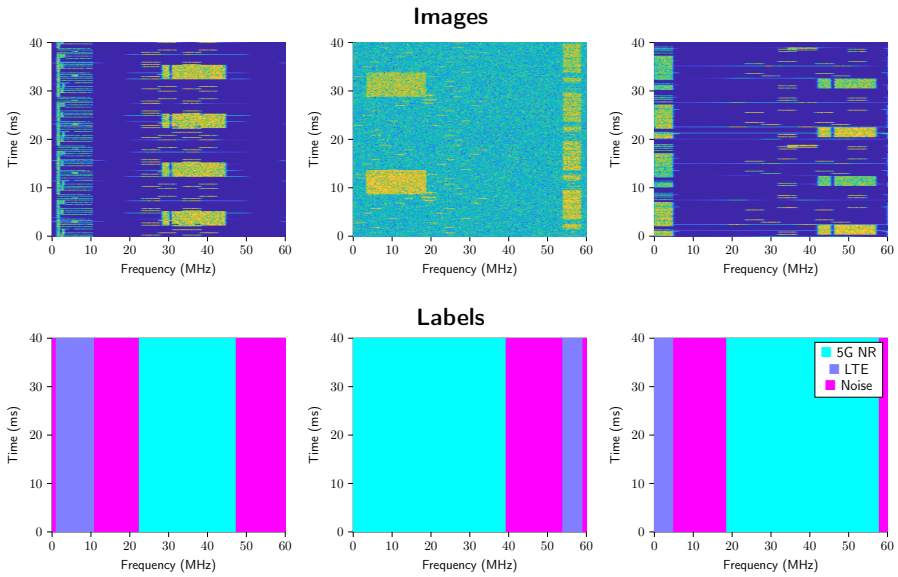

Figure 3.2: The raw signals and their corresponding label masks.

Matlab also provides a way to create synthetic data, by using the 5G- and LTE Toolbox, if more data is needed. The possible range of parameters can be seen in the table 3.1.

Table 3.1: The parameters of the 5G NR and LTE signals

Parameters	5G NR	LTE	Units
Bandwidth	[10, 15, 20, 25, 30, 40, 50]	[10, 5, 15, 20]	MHz
SNR	[40, 50, 100]	[40, 50, 100]	dB
Doppler	[0, 10, 500]	[0, 10, 500]	Hz
Sub-Carrier Spacing	[15, 30]	-	kHz
SSB Block Pattern	['Case A', 'Case B']	-	
SSB Period	20	-	ms
Duplex Mode	-	FDD	

The images included in the dataset were stored in PNG files, and the truth labels were stored in Hierarchical Data Format (HDF4) files. Because HDF4 is an older format and generally is not well supported for Python, the labels were converted to a newer format, HDF5. For that purpose, the tool h4toh5tools [18] was used.

3.2 Tools and materials

The experiments in FL were carried out in the Flower framework [19], which has the flexibility of setting up the experiments. The homomorphic encryption was done using Pyfhel [20], which has support for the CKKS scheme. During the training, functions from Scikit-learn [21] were used to calculate intersection over union (see 3.4.3), and F-score (3.4.4): The machine learning was done using the PyTorch library [22].

3.3 Solution Approach

3.3.1 Model Selection

Given the limited computational resources and large computational cost of machine learning and encryption, it is crucial to find the most optimal model for a given task. As of the time of writing, there are three available models for semantic segmentation tasks in PyTorch: DeepLabV3 [23], Fully-Convolutional

Network (FCN) [24] and LR-ASPP [25], each with a different backbone.

In order to find the right model, we split the data into train, test and validation sets in a 60:20:20 ratio. Next, we train every model for 10 epochs and test it on the validation set every iteration. The training performance can be seen in the figure 3.3.

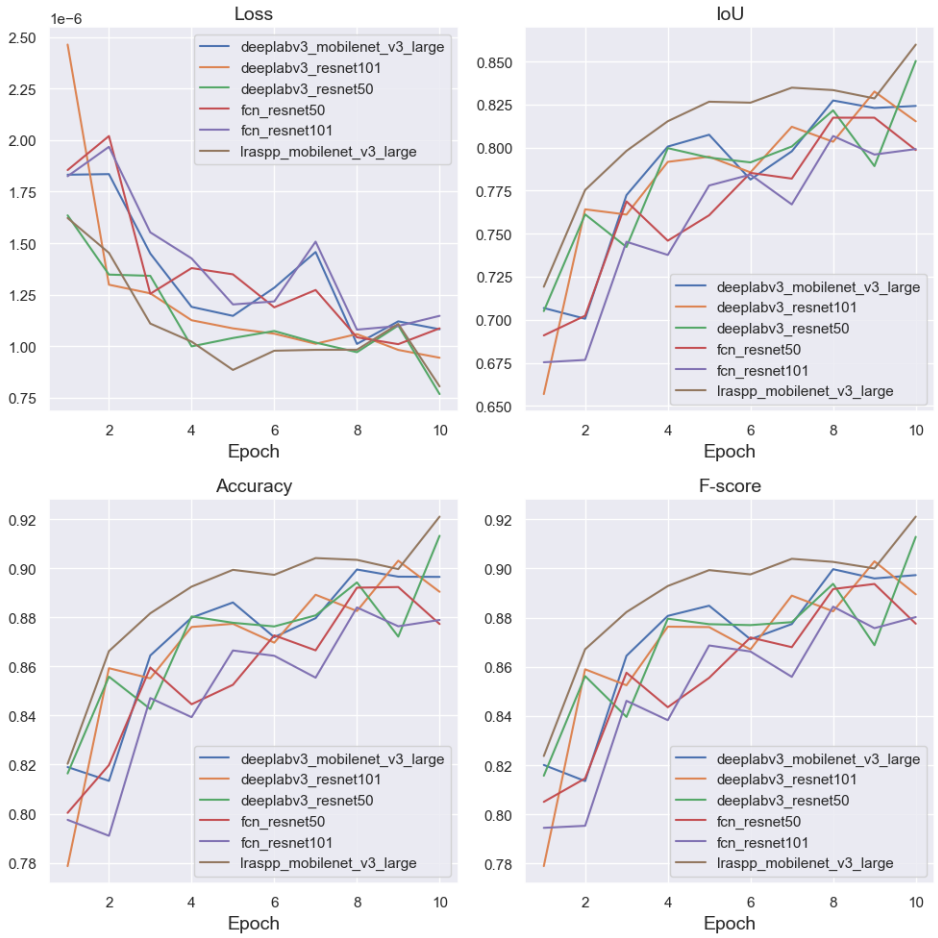

Figure 3.3: Model performance on the validation set.

Furthermore, we flatten the parameters and encrypt them using Pyfhel in batches of size 2^{14}. The final evaluation on the test set, the encryption times, and the model sizes are presented in 3.2.

Table 3.2: Performance of each model on the spectrum dataset.

Model	Params.	Enc. T.	Train T.	Loss	IoU	Acc.	F-Sc.
DL (MN)	11 M	15.58 s	193 s	1.34×10^{-6}	0.80	0.88	0.88
DL (RN101)	58.6 M	83.08 s	987 s	1.13×10^{-6}	0.79	0.87	0.87
DL (RN50)	39.6 M	55.58 s	711 s	$\mathbf{9.7 \times 10^{-7}}$	0.81	0.89	0.89
FCN (RN50)	32.9 M	45.61 s	528 s	1.25×10^{-6}	0.76	0.85	0.85
FCN (RN101)	51.9 M	72.14 s	804 s	1.21×10^{-6}	0.77	0.86	0.86
LR-ASPP (MN)	**3.2 M**	**4.45 s**	**144 s**	1.14×10^{-6}	**0.82**	**0.90**	**0.90**

Based on the results in figure 3.3 and table 3.2, the *LR-ASPP* model is chosen as it the smallest in size and achieves similar performance compared to the other models.

3.3.2 Pyfhel

The encryption is done with Pyfhel [20], an open source library for HE in Python, which uses Microsoft SEAL [26] as backend: a popular library for HE in C++ that supports the BFV [27] and the CKKS schemes.

To encrypt an object with Pyfhel, a Pyfhel object must first be initialized. This object can be configured with these parameters: polynomial degree n, $scale$, polynomial modulus qi_sizes, as well as security level sec.

The polynomial degree n determines the vector size that can be encrypted. In our scenario, given that we want to encrypt flattened parameters (which are long 1d vectors), we set this value as the highest currently possible 2^{15}. From experiments, we found that a lower value resulted in increased encryption time, due to the need to encrypt more batches for the same amount of parameters. The qi_sizes is an array of integers, each representing the bit size of primes is used for the modulus operations. For our initial, testing we set this value to $qi\_sizes = [60, 30, 30, 30, 60]$, which gives a total bit size of 210. The $scale$ is used for rescaling operations to reduce the noise.

We tested multiple configurations of the precision scale and polynomial modulus to measure the impact in the federated setting. For the security parameters, the available options are 128, 192 and 256 bit security.

3.3.3 Flower

We utilized the open-source Flower framework [19] for our experiments. This framework, designed for experimental research, is an easy-to-use API for FL. Figure 3.4 provides an simplified flow of the program with added encryption.

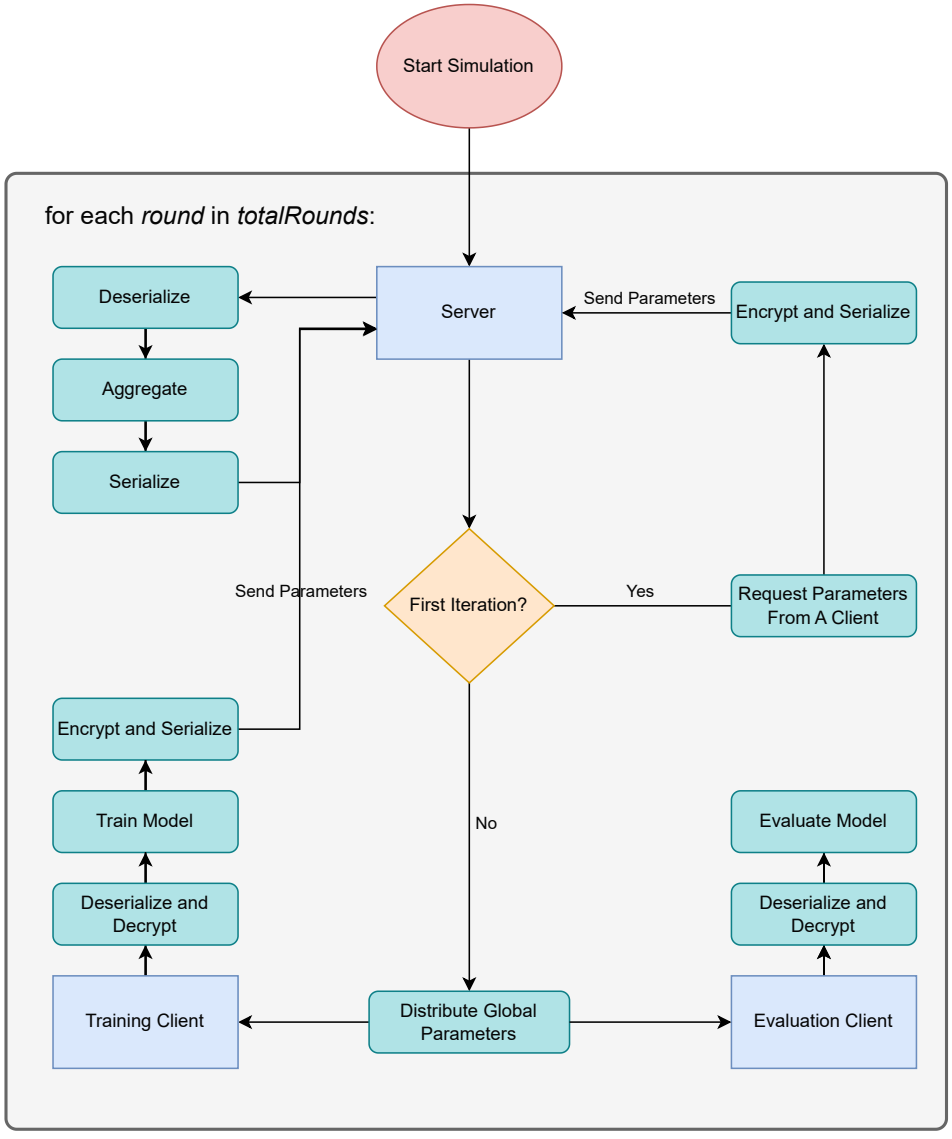

Figure 3.4: Simplified flow of the program in Flower with added encryption.

Starting off, we followed examples on Flowers website to set up the experiments. In general, the set-up consists of four main stages: data loading and partitioning, model creation, the Flower client setup and simulation parameters setup.

- **Data loading and partitioning**. The data is loaded, split to training and testing sets and partitioned to all clients. In our setup, we read the images stored in the 'dataset' folder, which contains two subfolders for images and labels. In the reading process we use a mapping function to convert the labels from 0, 127 and 255 to 2, 0, and 1 respectively. This step is needed because Pytorch's Cross Entropy Loss function expects the labels to contain indices in range [0,C) where C is the number of classes. Without this mapping, the function might interpret it as a 255-class, and not a 3 class problem, yielding wrong results. In the preprocessing step, we also perform data normalization to scale the input data from [0, 255] to [-1, 1].
- **Model creation**. We load the 3.4 million parameter model, LR-ASPP (MobileNetV3) for every client as it was shown to provide the best results in section 3.3.1, provided by Pytorch.
- **The Flower client**. There are two types of clients in Flower: NumPyClient and Client. The main difference is that the NumPyClient takes care the of the serialization and deserialization itself, but in Client that can be customized. However, our experiments showed that NumPyClient would return an error when trying to serialize an encrypted Pyfhel object. Therefore, a decision was made to use Client instead.
- **Parameters**. To start the simulations, we used Flower's $start\_simulation$ function which requires the following input: a client function, the number of clients, configurations (number of rounds), a strategy and client resources. The client function is used to initialize the client objects. Here, we specify the dataset partition for each client, the model and a boolean value used to determine if client should use HE. In each experiment we used a total of 10 clients, and a sample pool of 5 evaluation clients and 2, 5 or 7 training clients.

3.3.4 Strategy

The strategy was FedAvg both for the unencrypted and encrypted scenarios. However, we changed the aggregation function to accommodate for the en-

crypted encrypted case.

3.3.5 Weighting function

Flower allows specifying a weighting function for either a distributed evaluation on client side or on the server side. Since the server does not have access to the unencrypted parameters, we use distributed evaluation only. The distributed function allows finding a weighted average of evaluation results, thus giving more accurate results. Here we used a sample of 5 evaluation clients.

3.3.6 Serialization and Deserialization

To send the parameters using Client, they need to be stored in Flowers own Parameters object. Here, we also did serialization using Pyfhels $to\_bytes()$ method. The serialization function is given by:

Algorithm 1 Serialize function

function SERIALIZE($encryptedParameters$)
    $batches \leftarrow []$
    for each $batch$ in $encryptedParameters$ **do**
        $bytes \leftarrow batch.to\_bytes()$
        append $bytes$ to $batches$
    end for
    return $Parameters(tensors = batches, tensorType =' PYCtxt')$
end function

Once the parameters object is received, it needs to be deserialized. In Pyfhel, this can be done with converting the stream of bytes back into a PyCtxt object, together with a Pyfhel object. The deserialize function is given as follows:

Algorithm 2 Deserialize function

function DESERIALIZE($parameterObject$, HE)
    $output \leftarrow []$
    for each $batch$ in $parametersObject.tensors$ **do**
        append $PyCtxt(bytestring = batch, pyfhel = HE)$ to $output$
    end for
    return $output$
end function

3.3.7 Encryption

The encryption function is used by the training clients and can be summarized as follows. The function takes the flattened parameters as input and divides them into batches defined of size defined by the Pyfhel object. The batches are stored in a list and returned. The batch size is determined by the polynomial modulus degree parameter of Pyfhel and is always $n/2$, as determined by the $HE.get\_nslots()$ method.

Algorithm 3 Encryption function

function ENCRYPT($flatParameters, HE$) ▷ HE: Pyfhel object
    $B \leftarrow HE.get\_nSlots()$ ▷ B: Batch size
    $N \leftarrow ceil(length(flatParameters)/batchSize)$ ▷ N: # of batches
    $Res \leftarrow []$ ▷ Res: Result list
    for $i \leftarrow 0$ to N **do**
        $a \leftarrow i * B$
        $b \leftarrow (i + 1) * B$
        $Batch \leftarrow flatParameters[a : b]$
        $encryptedBatch \leftarrow HE.encryptFrac(Batch)$
        $Res.append(encryptedBatch)$
    end for
    return Res
end function

3.3.8 Aggregation

The aggregation function is used by the server. It can be summarized as follows. It takes a list containing sub-lists of parameters. It uses two loops. The first one adds the parameters of each client to a single list, as seen in lines 6 to 10. The second one finds the average by dividing the list by the number of clients. Since the division is not possible with the CKKS scheme, we find a fraction number by dividing 1 by the number of clients and multiply the results.

Algorithm 4 Aggregation function

```
1:  function AGGREGATE(paramsLists, HE)
2:      C ← length(paramsLists)                    ▷ C: Number of clients
3:      P ← length(paramsLists[0])                 ▷ P: Number of parameters
4:      N ← HE.encode(1.0/C)
5:      aggParams = paramsLists[0]
6:      for each params in paramsLists[1 :] do
7:          for i ← 0 in params do
8:              aggParams[i] ← aggParams[i] + params ▷ Calculating sum
9:          end for
10:     end for
11:     for i ← 0 to P do
12:         aggParams[i] ← aggParams[i] * N          ▷ Calculating average
13:     end for
14:     return aggParams
15: end function
```

3.3.9 Decryption

The decryption function is called on the client side and is the one of the last steps of the FL round. This function consists of two steps: the initial decryption of parameters and reshaping these parameters back to their original form. The original shapes of the parameters, obtained from the flattening operation, are stored in the clients class object. The first step happens from line 4 to 7 in the algorithm. For every encrypted parameter, the HE.decryptFrac function is called. The resulting decrypted parameters are then stored in a list. The following step, can be seen in the line 12 to 18. Here is where the reshaping is done. For that purpose Numpy's reshape function is used. Once complete,

the decrypted parameters can be then combined together to the model's state dictionary to update the model and conclude one round of the FL process.

Algorithm 5 Decryption function

```
function DECRYPT(encryptedParams, HE, originalShapes)
    flatParams ← []
    N ← length(encryptedParameters)
    for i ← 0 to N do
        decParams ← HE.decryptFrac(encryptedParams[i])
        extend flatParams by decParams
    end for
    M ← sum(product(dimensions in originalShapes))
    flatParams ← flatParams[: M]        ▷ Remove padding caused by encryption
    decParams ← []
    i ← 0
    for each shape in originalShapes do
        D ← product(shape)
        batch ← flatParams[i : i + D]
        batch ← batch.reshape(shape)
        append batch to decParams
        i ← i + D
    end for
    return decParams
end function
```

3.4 Evaluation metrics

3.4.1 Loss

Loss is a measurement of how well the model makes predictions compared to the actual data. It can be used both in training and validation. In training, the models main objective to minimize the loss by adjusting the parameters by comparing the prediction output and the actual data [28]. When used in validation data, it is used as a measure of how well the model generalizes on unseen data.

In this project, we use one specific type of loss know as called cross-entropy loss. This type of loss measures the difference between the *probability distributions* of the model's output and the target labels. For a given sample n the cross-entropy loss can be calculated as follows [29]:

$$l_n = -w_{y_n} \log \frac{exp(x_{n,y_n})}{\sum_{i=1}^{C} \exp(x_{n,c})} \cdot y_n \tag{3.1}$$

where x is the input, y is the target, w_c is a weighting factor, C is the number of classes, N is the mini-batch dimension.

3.4.2 Accuracy

For an semantic segmentation problem, an important evaluation metric is accuracy. Accuracy is a measure of how many points are classified as correct out of the total number of points. More formally it can be given as

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.2}$$

where TP is true positive, TN is true negative, FP is false positive, FN is false negative [30].

One limitation of accuracy is that it does not take into the account the imbalance of the datasets. If the class distribution is heavily skewed towards one class, that can lead to a high accuracy score, despite doing bad predictions on the minority classes.

3.4.3 Intersection over Union

Intersection over Union (IoU) or also known as Jaccard index, is another commonly used metric in computer vision problems. The IoU is defined as the size of the intersection divided by the size of the union between two sets [31]. This can be more formally written as:

$$IoU = \frac{|A \cap B|}{|A \cup B|} \tag{3.3}$$

for two sets A and B.

In practice, the set A usually represents the input images (input pixels), while B represents the prediction output (output pixels). The numerator thus finds the area of overlap between the input and the output. The denominator calculates the total area. The division operation results in a score that $0 \leq IoU \leq 1$. An IoU that is close to 1 represents a large overlap and good prediction, while an IoU close to 0 represents small overlap and bad prediction.

3.4.4 F-Score

The F-score is a combination of two other metrics, precision and recall. The formula for it is given by [32]:

$$\text{F-score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{3.4}$$

Precision is a measure of the correct predictions out of all the positive predictions by the model:

$$\text{precision} = \frac{TP}{TP + FP} \tag{3.5}$$

Recall is a measure of the correct classifications on the positive values:

$$\text{recall} = \frac{TP}{TP + TN} \tag{3.6}$$

By using the formula in 3.4, we find a balance between the precision and recall.

3.5 Limitations and Challenges

The biggest challenge encountered from the experiments was the memory usage. The experiments were done on a single PC with 32 GB of RAM, this meant that the number of clients that could be simulated was limited.

For instance, when doing experiments with 10 clients and a high amount of rounds, the program would use up all memory at round at some point and crash. The addition of encryption made the problem worse.

This led to a reduction in the number clients being simulated. From testing we found the highest possible was seven clients, both for the unencrypted and encrypted case.

Chapter 4

Results

4.1 Baseline model

First, we test the models performance in a centralized setting. Initially, we split the data into a training, validation and testing set in a 60:20:20 ratio. Then, the model is trained with early stopping and patience level of 5. The results show that the model converges early and stops just after 16 epochs. The evaluation on the test set after training the model show gives the following metrics: Loss $= 5.83 \times 10^{-7}$, IoU $= 0.83$, Accuracy $= 0.90$, F-score $= 0.90$. Figure 4.1 shows the evaluation metrics on the validation set after each iteration. 3.2 illustrates the model predictions.

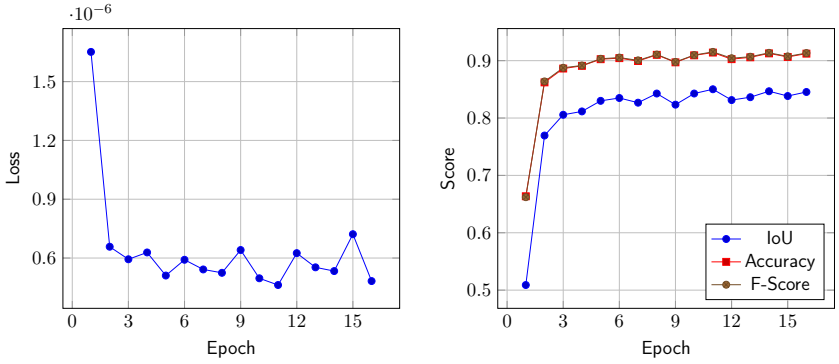

Figure 4.1: Early convergence of validation metrics after only a few rounds.

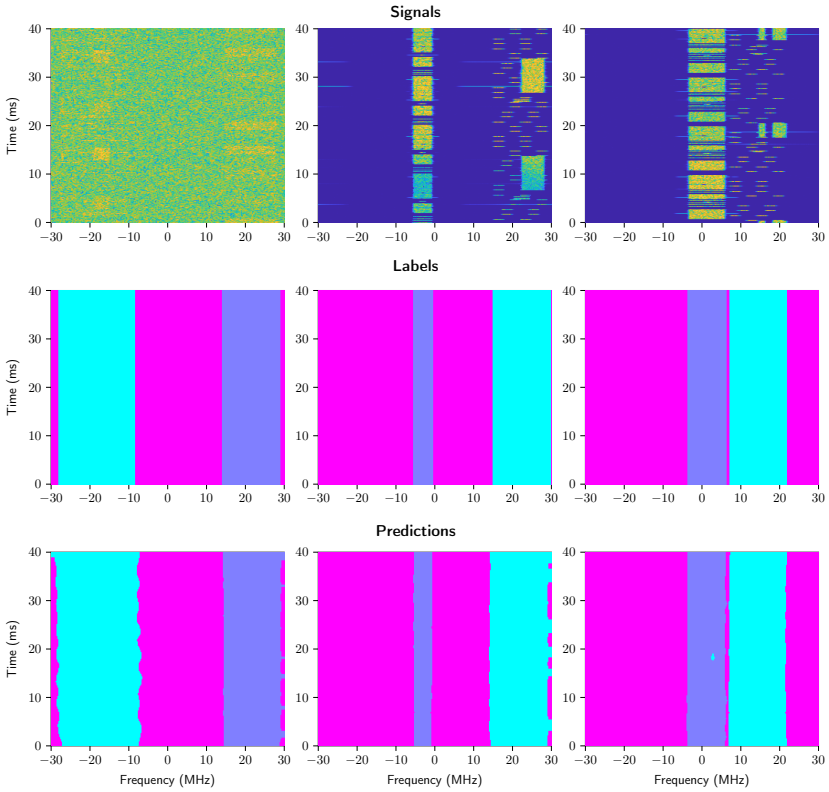

Figure 4.2: Top row shows the wireless signals. Middle row shows the ground-truth labels, and the bottom row shows the labels predicted by the model.

4.2 FL Without Encryption

For our second experiment, we validate the model performance in the distributed setting without using encryption. We do three different tests, each done with either 2, 5, or 7 clients. In all cases, the tests are configured to run for 100 iterations, with one local update per iteration. Figure 4.3 provides the validation metrics for each round. A quick look at the figure shows that the model converges around the 20 round mark for all metrics.

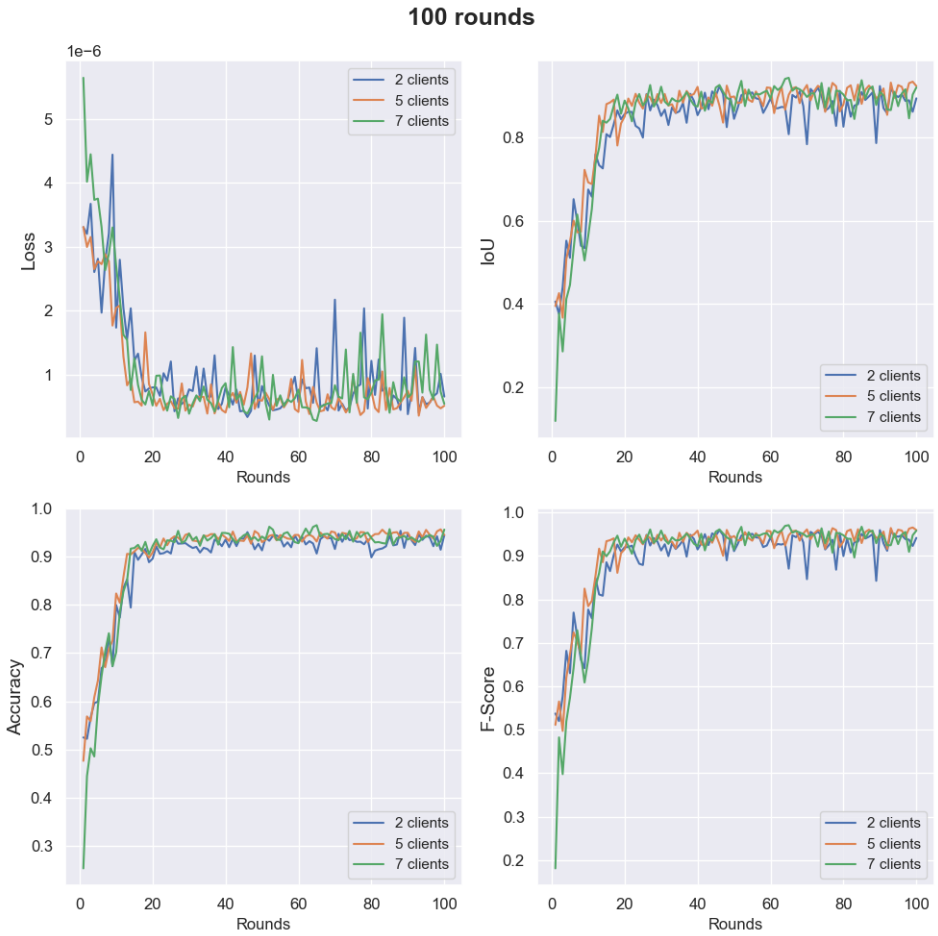

Figure 4.3: Results from plain FL, showing convergence after 20 rounds.

4.3 FL With Encryption

In the next experiment we reduced the amount of training rounds from 100 to 30, and re-run the experiments with and without encryption using 7 clients. For the encrypted case, we set the initial parameters accordingly: $n = 2^{15}$, $scale = 2^{30}$, $qi\_sizes = [60, 30, 30, 30, 60]$ (210 bits) and $sec = 128$ bits. The performance results can be seen in the figure 4.4. The figure shows that model performance during the training process is comparable to the plain model.

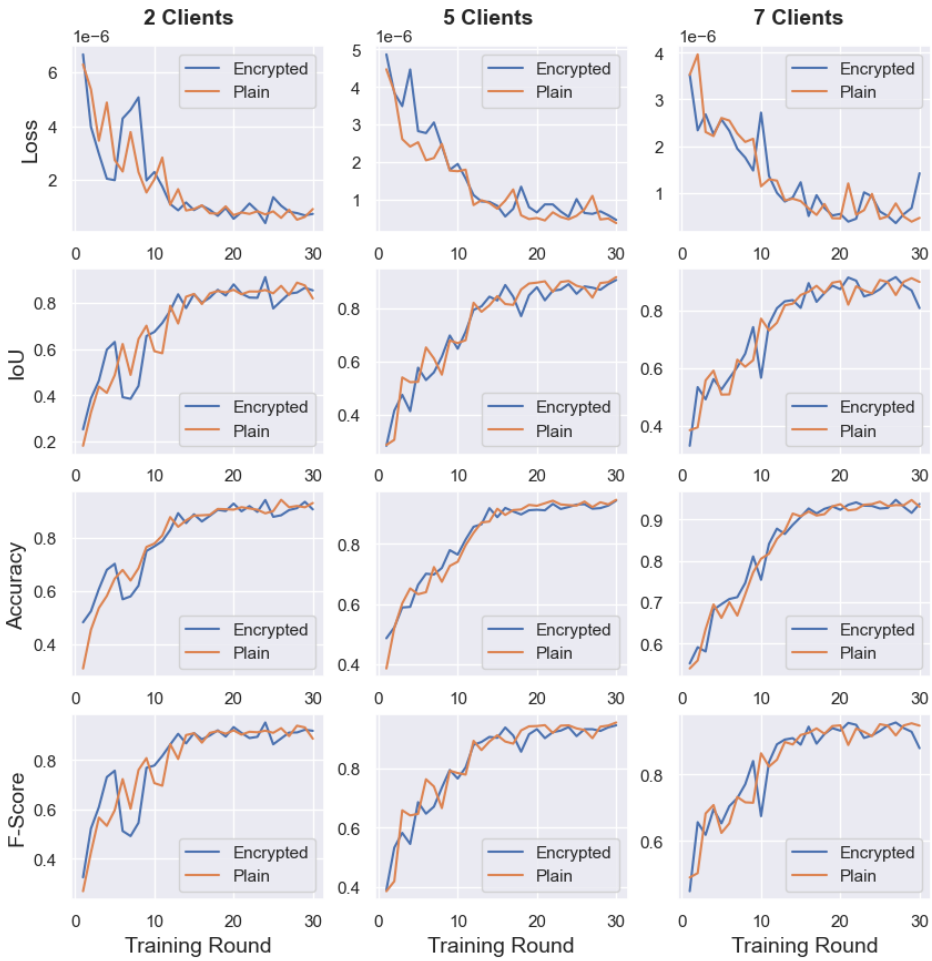

Figure 4.4: FL with and without encryption, showing similar results for each metric.

There is, however, a noticeable difference in runtime. The experiment's duration was 1636, 2514, 3353 seconds for 2, 5 and 7 clients respectively for the encrypted case. This is a large increase compared to the plain model where the experiment's duration was 311, 402 and 652 seconds for 2, 5 and 7. Figure 4.5 shows the time distribution of the following tasks: encryption, decryption, aggregation, evaluation, training. A quick look at the figure confirms that encryption and decryption account for the majority of the time spent.

Figure 4.5: The time distribution of clients using homomorphic encryption.

4.4 Security levels

This section explores the impact on performance of varying encryption security levels. We test for the three available security levels: 128, 192 and 256 bits, with 7 clients each for each test. We also lower the qi_sizes parameter to a total of 150 bits. The results can be seen in figure 4.6.

Interestingly, we observe no noticeable impact in terms of performance between the three security levels. Each experiment had similar runtime, at around 2100 seconds each. However, given that the last experiment, which also included a test for 128 bit security, had a runtime of 3353 seconds, we can infer that the change was a result of reducing the qi_sizes parameter.

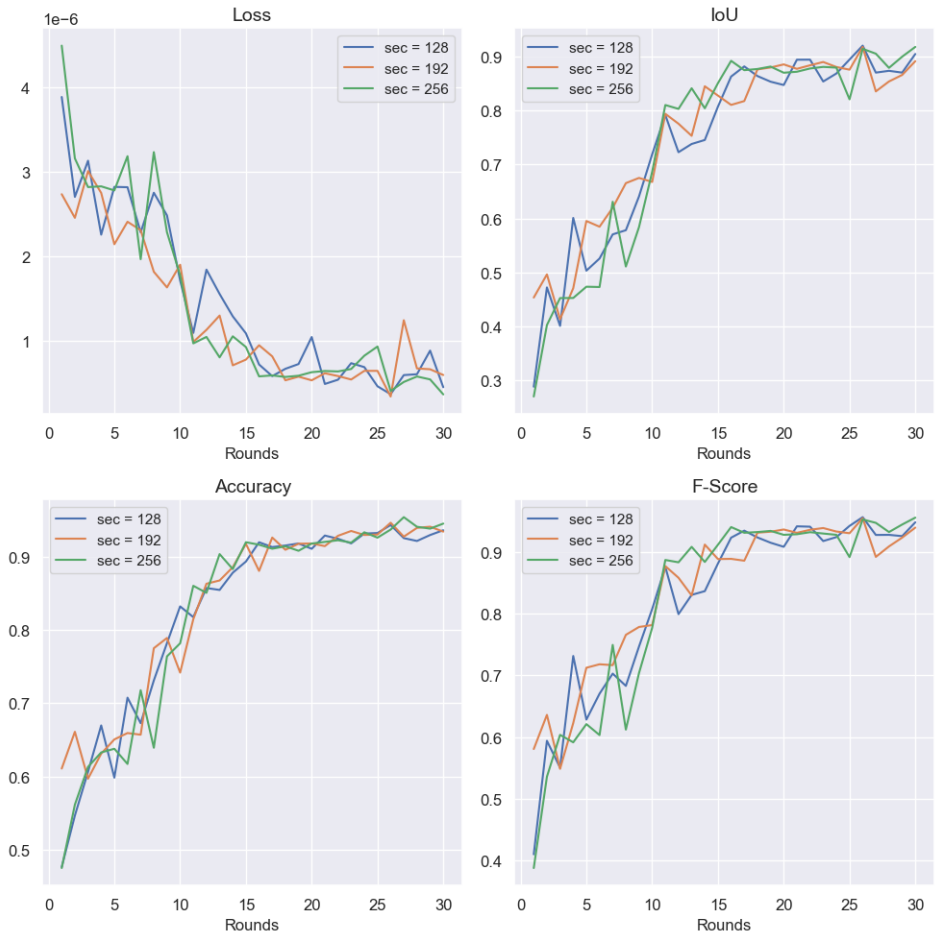

Figure 4.6: Performance for 7 clients with 3 levels of security.

4.5 Scale parameter

In an effort to reduce computational overhead, we conducted experiments with varying the scale parameter. The polynomial modulus was fixed at 120 bits, while the scale values were tested in the range from 2^{20} to 2^{30}. The results are presented in figure 4.7.

Interestingly, the highest scale value at which the model converged was 2^{25}, although a sharp decrease can be observed at the end of the training cycle. As

such, 2^{26} appears to be the smallest viable scale for the model. The experiments also showed that the scale parameter had minimal impact on runtime; all experiments ran for around 1630 seconds for 5 clients.

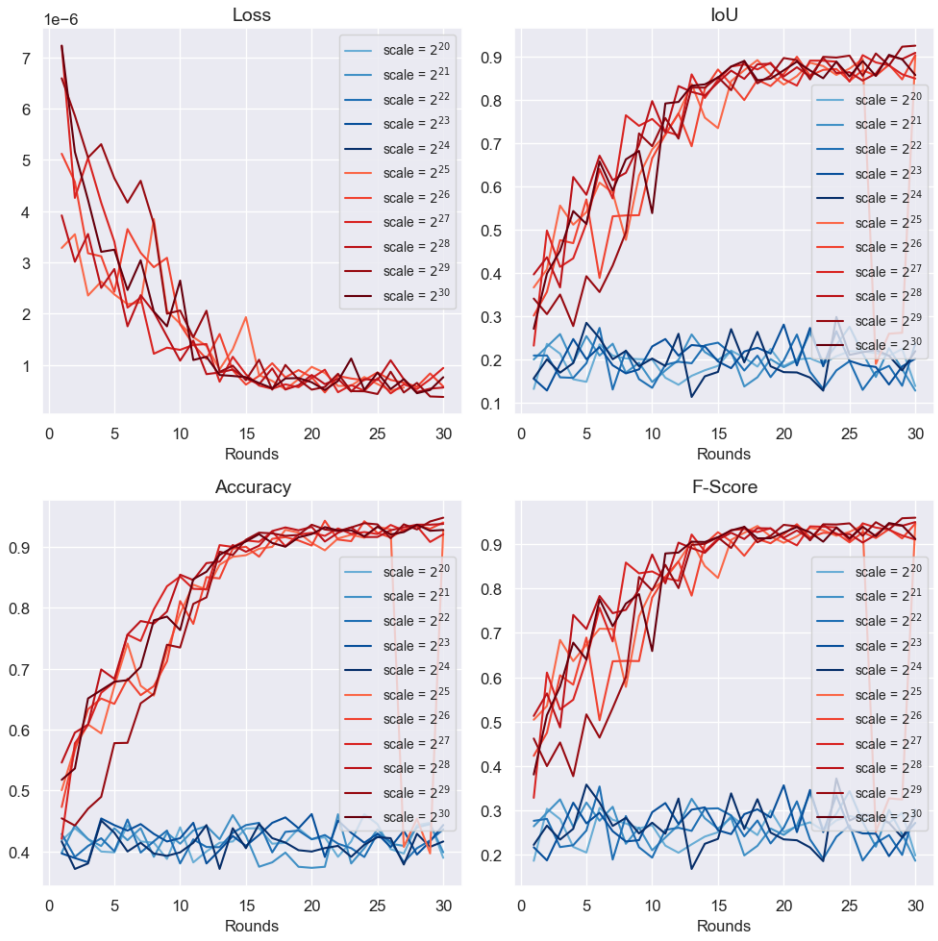

Figure 4.7: Impact of the scale parameter on models performance. A threshold at 2^{25} can be observed.

Chapter 5

Discussion

The results from figure 4.4 show that there is no noticeable difference in terms of performance between the plain and the encrypted model. The noise added from encryption is not enough to affect training in any meaningful way.

A big difference can be seen however in the runtime of the experiments. The added encryption has caused an increase of 587 %, 624 % and 679 % for 2, 5, 7 clients respectively. As seen from figure 4.5, the majority of increase is caused by the added computational overhead from encryption and decryption on the client side. However, the encrypted aggregation could have an impact as well.

Surprisingly, a significant reduction in runtime was seen from reducing the modulus parameter from 210 to 150 bits. However, no significant change was seen after reducing this parameter any further.

We infer that training a distributed model can indeed be done securely with help of FL and HE to gain the benefits of CNN-based spectrum sensing, however the computational cost of that needs to be carefully considered.

In this project we studied how the parameters affect the models computational cost. However, more future work can be done to find the effect of using a higher modulus size and scale. Furthermore, the experiments in this project were conducted with a small amount of clients due to memory constraints. More work can be done on a bigger scale with larger amount of clients using a bigger dataset.

Chapter 6

Conclusion

Machine learning (ML) techniques can serve a great benefit to the traditional way of doing spectrum sensing. However, the large amount of data which is needed to make these models reliable, and the nature of the spectrum sensing data requires that such systems are designed in a secure and reliable manner in order to not compromise users' privacy.

In this project this problem is addressed by using FL, enabling a distributed model to be trained without needing to share any client data to a central server. As additional layer of security, HE was added to make inference attacks on the models parameters impossible.

We did an evaluation of the LR-ASPP model in the federated learning setting using 2, 5 and 7 clients and measured the results. A convergence of the model after 20 rounds was found, which we utilized in the following experiments.

Furthermore, we did a comparison between an unencrypted and encrypted model, showing no impact in terms of prediction performance, but a large increase increase in computational overhead caused by the encryption and decryption process.

Next, we examined the correlation between performance and key sizes of 128, 192 and 256 bits. We found no significant impact of the key size on performance. A significant change was noticed that can be attributed to reducing the qi_size parameter from 210 to 150.

Lastly, we reduced the qi_size further to 120 bits and tested the scale parameter. We found a threshold of 2^{25} where the model can be trained.

In this project, we demonstrated that the CKKS scheme is a viable option for distributed spectrum sensing with FL. We found minimal impact on prediction performance between the plain and encrypted model. However, the increased computational cost introduced by encryption, requires careful consideration to determine if it feasible in real-world applications.

Bibliography

[1] F. Hu, B. Chen, and K. Zhu, "Full spectrum sharing in cognitive radio networks toward 5g: A survey," *IEEE Access*, vol. 6, pp. 15 754–15 776, 2018.

[2] J. Gao, X. Yi, C. Zhong, X. Chen, and Z. Zhang, "Deep learning for spectrum sensing," *IEEE Wireless Communications Letters*, vol. 8, no. 6, pp. 1727–1730, 2019.

[3] F. O. Catak, M. Kuzlu, S. Sarp, E. Catak, and U. Cali, "Mitigating attacks on artificial intelligence-based spectrum sensing for cellular network signals," in *2022 IEEE Globecom Workshops (GC Wkshps)*, 2022, pp. 1371–1376.

[4] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, A. Singh and J. Zhu, Eds., vol. 54. PMLR, 20–22 Apr 2017, pp. 1273–1282. [Online]. Available: https://proceedings.mlr.press/v54/mcmahan17a.html

[5] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 691–706.

[6] J. Park and H. Lim, "Privacy-preserving federated learning using homomorphic encryption," *Applied Sciences*, vol. 12, no. 2, p. 734, 2022.

[7] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "{BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning," in

2020 USENIX annual technical conference (USENIX ATC 20), 2020, pp. 493–506.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.

[9] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (Csur)*, vol. 51, no. 4, pp. 1–35, 2018.

[10] R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[11] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[12] Inferati, "Introduction to the ckks/heaan fhe scheme," 2022. [Online]. Available: https://inferati.azureedge.net/docs/inferati-fhe-ckks.pdf

[13] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," Dec. 2015, arXiv:1511.08458 [cs]. [Online]. Available: http://arxiv.org/abs/1511.08458

[14] MathWorks, "Object Detection." [Online]. Available: https://mathworks.com/help/vision/object-detection.html

[15] ——, "Semantic Segmentation." [Online]. Available: https://mathworks.com/help/vision/semantic-segmentation.html

[16] Y. Zeng, Y.-C. Liang, A. T. Hoang, and R. Zhang, "A review on spectrum sensing for cognitive radio: challenges and solutions," *EURASIP journal on advances in signal processing*, vol. 2010, pp. 1–15, 2010.

[17] MathWorks, "Spectrum Sensing with Deep Learning to Identify 5G and LTE Signals." [Online]. Available: https://mathworks.com/help/comm/ug/spectrum-sensing-with-deep-learning-to-identify-5g-and-lte-signals.html

[18] T. H. Group, "h4toh5tools," 2023. [Online]. Available: https://portal.hdfgroup.org/display/support/h4h5tools

[19] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, H. L. Kwing, T. Parcollet, P. P. d. Gusmão, and N. D. Lane, "Flower: A Friendly Federated Learning Research Framework," *arXiv preprint arXiv:2007.14390*, 2020.

[20] A. Ibarrondo and A. Viand, "Pyfhel: Python for homomorphic encryption libraries," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2021, pp. 11–16.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[23] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking Atrous Convolution for Semantic Image Segmentation," *CoRR*, vol. abs/1706.05587, 2017, arXiv: 1706.05587. [Online]. Available: http://arxiv.org/abs/1706.05587

[24] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *CoRR*, vol. abs/1411.4038, 2014, arXiv: 1411.4038. [Online]. Available: http://arxiv.org/abs/1411.4038

[25] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for MobileNetV3," Nov. 2019, arXiv:1905.02244 [cs]. [Online]. Available: http://arxiv.org/abs/1905.02244

[26] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[27] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[28] Google, "Descending into ML: Training and Loss." [Online]. Available: https://developers.google.com/machine-learning

[29] Pytorch, "CrossEntropyLoss," 2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[30] PyTorch, "Accuracy," 2023. [Online]. Available: https://pytorch.org/ignite/generated/ignite.metrics.Accuracy.html

[31] scikit learn, "sklearn.metrics.jaccard_score." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.jaccard_score.html

[32] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International journal of data mining & knowledge management process*, vol. 5, no. 2, p. 1, 2015.

Appendix A

Additional Information

A.1 GitHub repository

The code for this project can be found in the link below:

`www.github.com/Empie/5G-Networks-Privacy`