Making a secure framework for Federated Learning

Vaibhav Gusain B.Tech

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Aljosa Smolic

September 2021

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Vaibhav Gusain

August 30, 2021

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Vaibhav Gusain

August 30, 2021

Acknowledgments

I'd like to take this opportunity to acknowledge Prof. Aljosa Smolic and Dr. Cagri Ozcinar for continued support and guidance throughout the process. I have been extremely lucky to have my supervisors who provided insightful discussions about the research and who responded to my questions and queries so promptly despite their busy agenda. Thanks to there amazing guidance, perfect time management, endless encouragement, and consistently wise advice. They were a true source of inspiration. Finally, I'd like to express my gratitude towards my family and friends for their constant encouragement and for helping me edit. I wouldn't have been able to complete this journey if it weren't for all the motivation from everyone.

Vaibhav Gusain

University of Dublin, Trinity College
September 2021

Making a secure framework for Federated Learning

Vaibhav Gusain, Master of Science in Computer Science

University of Dublin, Trinity College, 2021

Supervisor: Aljosa Smolic

Federated learning approach was built so that we can use data which is distributed at different user devices to train a machine learning algorithm, without the data being actually transferred outside the user device. Thus facilitating the learning of the model and also not hampering the user privacy-the user data never leaves the local device. However recent approaches have shown that private user data can be exploited by using the gradients or the weights that the edge model share with the main server. In this dissertation, we would like to introduce an agent which would allow the user devices to share the weights in an encrypted way, the server would do its update on such encrypted weights and then return the updated weights to the user

Summary

Neural networks are data driven model i.e they require a lots of data to train and work properly. However with the rising use of neural network there are a lot of concern about the security regarding the private information present in those data. To mitigate such problems federated learning were introduced, but research still shows that user data can still be exploited in that approach.

In this research we will first show how the traditional way of training the neural network poses a threat to user private information. Then we will briefly talk about federated learning and how we can still exploit the information in such a setting. We will also demonstrate how we can exploit the data from a federated learning setting. Then we will introduce our secure federated learning framework where we will first show that it not only gives almost equal accuracy as that of the federated learning framework but also provide an added layer of security so that its really hard to exploit user data in such a setting.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Security and privacy both go hand in hand. If our system is secure then we can make sure that the user privacy is preserved in our system. On the other hand if our system is not secure the user privacy might be at risk and can easily be compromised by a malicious attacker. Machine learning models are data driven models. These models are trained on raw data and more data we have more accurate model we can build. Most of the times such data often contain user private information which if fallen into the hands of attacker can be misused in anyway the attacker sees fit.
Traditionally we would collect all of the data at one place to train our machine learning model, this approach was very risky as we were putting all of the user data at one place thus making it easier for an attacker to steal the information. To overcome this authors at [1] developed a new algorithm called Federated learning where instead of collecting the user data to a single storage device on the cloud we would only collect the model updates. Thus user data never leaves there device and thus making it harder to steal such data.But authors of [2] [3] quickly demonstrated that we can still steal user information in such a setting if no encryption is used.
By looking at the severity of the problem we decided to research and develop a secure framework for the federated learning algorithm. In this work we will try to describe our secure framework for the federated learning setup. We will start by talking about our motivation and the literature review done. Then we will discuss about the framework

we have built followed by the results and conclusion.

1.2 Motivation

With the advances in the field of deep learning, we are at a stage where we can train neural networks to automate certain tasks. In some areas the accuracy of a neural network surpasses the accuracy of an average human, for example: Facial recognition systems. To get such an accuracy neural networks are required to be trained on huge amounts of user data. Traditionally, to train such networks, the data is collected over a single storage unit and then the neural network is trained on a server using the collected data.

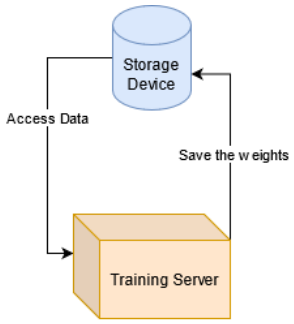

Figure 1.1: Figure shows traditional way of training neural network

Figure (1.1) shows how traditionally neural networks were trained. As evident from the image the data is first stored into a single storage device, that device is connected to a high processing CPU which trains the neural network and saves back the results in the storage device. Such an approach however efficient and was easy to scale but possessed a crucial security threat for user data.

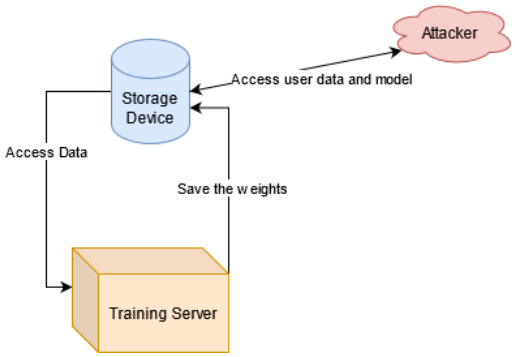

Figure 1.2: Figure shows how easy it is to attack traditional setup

Recent researches [1] [4], have pointed out that such an approach of training poses a crucial security threat. Since we are collecting all of the user information at a single location on a server, if the security of the training machine is compromised, then the hacker will not only have the AI model we have created but might also get access to user sensitive information that was being used to train our neural network. This is evident from figure (1.2) we can see that if an attacker got the access to our storage device, then they will not only have our trained neural network but it will also have the access to the training data which might contain user private information. Thus putting user private information at a risk.

Figure 1.3: Figure shows in federated learning only the model updates are shared and the user data never leaves the edge device.

To mitigate the issue, federated learning was introduced by [1] in which a user was able to train neural network models without having to collect all of the user data at a

single location. In such a setting each device receives the copy of the global neural network and fine tune it on its own training data. Once the training is done it shares the trained weights/gradients with the server, then the server aggregates all the collected weights thus updating the global model, it then shares back the new global model to all the devices. Hence even if there is a security breach, the hacker will only have access to the neural network and not the user data thus preserving the privacy of the user. Figure (1.3) shows a traditional federated learning setup. From the figure we can see how the local data of the user never leaves the edge device but only the model updates are shared with the server thus making it harder for an attacker to steal the raw user data. .

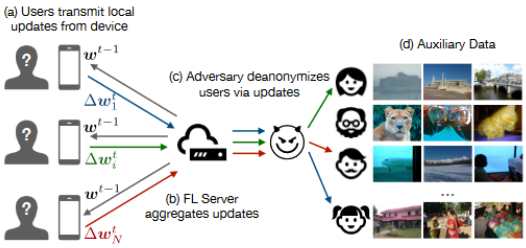

Figure 1.4: Figure demonstrating that it is still possible to extract user information from the shared weights

But research done by [3][2] suggests that, it is possible to get the user information in the federated training approach using the weights/gradient shared by the devices to the server, i.e., if the security of the server is compromised or if it is a malicious server then it might extract the user information using the shared weights. Thus if the shared weights are not encrypted then the attacker might still get the private information of the user. From figure(1.4) we can see how the attacker was able to reconstruct the user images which they used to train their neural networks just from the shared weights. Which thus again goes on to prove that if the shared weights of the networks are not encrypted then the user private information can still be reconstructed from the shared weights thus putting user privacy at a risk.
However there is a handful of research that is being done on how we can encrypt the data before we feed it into the models. In such a technique usually the input to a model is distorted by using some predefined distortion algorithm which encrypts the input

in such a way so that image is not recognizable by the naked human eye and is sent to the neural network. In such cases even if the security of the model is compromised the hacker won't be able to access the user information as the images it will get will only be distorted images. However in most of such cases we need to retrain our neural network using the new distorted images, so that it can learn what features to learn from such new images. Which again requires a lot of time and effort. Recent research done by the authors at [5] mitigate this issue, as they present an encryption algorithm where we can use our existing convolutional models and encrypt both the images and the neural network in such a way so that we still get the desired result without having to retrain our model and still ensuring that the privacy of the user data is preserved. This opens a lot of possibilities as in this encryption technique the pre-processing time is negligible and we don't need to retrain our neural network for the new encrypted images.

1.3 Reader's Guide

In this research we are trying to propose a federated learning approach, which combines an already existing federated learning algorithm and an encryption algorithm, which would let researchers train the models in an already federated learning environment, but with the added features of model encryption. So that even if the security of the server is compromised or if there is a malicious machine in the federated learning environment the sensitive user information will not be revealed. The format of this paper is as follows, we will first talk about the related research in the field, then we will describe the methodology used by us and how the data was processed. Then we will end the thesis showing our results and giving a conclusion and scope for future work.

Chapter 2

Background Related Work

2.1 Federated Learning

Traditionally neural networks are trained using big GPU clusters, all the data is first transferred to a single storage location and then used for training. However, while this approach has worked over the past years, this has some issues:

1. If the security of the server device is compromised then the privacy of the data is also compromised.

2. Such approach can not be used to train models that require more human interactions i.e training a model for the word prediction for a virtual keyboard software.

To tackle the problems stated above authors of [1], came up with the idea of federated learning while working for Google in 2017. In federated learning, instead of using a big GPU cluster to train the machine learning model, all the devices that are participating in the training first downloads the current model from the cloud, do a small focused update on the current model using the data present on the device then share the updated model again with the cloud. The cloud will then aggregate the results received by different devices and will update a new model which devices can download and use. By having such an approach the devices don't need to share there private information but rather can share the model updates with the server, thus ensuring the user privacy.

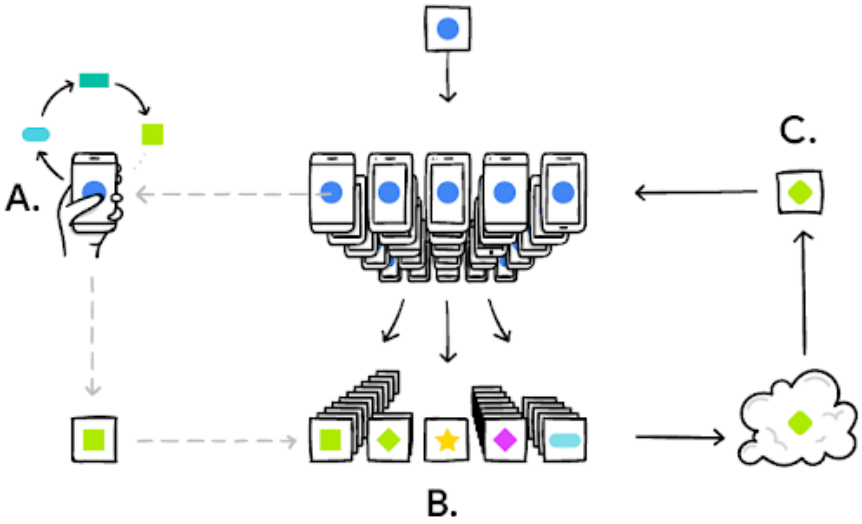

Figure 2.1: Figure shows how federated learning is done. Here the model is first trained on the local device A. Then updates from all such devices are collected at B. Then an update on the server side is done and the new weights are shared again at C to all the other devices

As shown in the Algorithm 1 every client first receives the global copy w_0 from the server and then it runs n number of local epochs on its own data. Then the server picks a random set of clients, these will be the clients that will participating in the current learning cycle. Then those selected clients do a local update on there device using there own private data for E number of epochs. Once they have completed there local updates, they share their weights/ gradients with the server. Once the server receives updates from all the selected clients, it takes a weighted average of all the weights and update the current weight w_0 with the new updated weights and share the new weights with the clients.
Note here C is the fraction of clients of clients involved in current federated learning cycle. i.e if $C = 0.5$ then only 50% of the client will be involved in the current learning cycle. However for the experiments the authors used C=1.0 i.e all of the clients were involved in the learning cycle.

Algorithm 1 FederatedAveraging - The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and is η is the learning rate

Server executes :
initialize w_0
for each round t=1,2,... **do**
  $m \leftarrow \max(C.K, 1)$
  $S_t \leftarrow (random\ set\ of\ m\ clients)$
  for each client $\in S_t$ in parallel **do**
    ${w_{t+1}}^k \leftarrow$ ClientUpdate(k,w_t)
  end for
  $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} {w_{t+1}}^k$
end for
ClientUpdate(k,w) : // Run on client k
$\beta \leftarrow (split P_k\ into\ batches\ of\ size\ \beta)$
for each local epoch i from 1 to E **do**
  for *batch* $b \in \beta$ **do**
    $w \leftarrow w - \eta \bigtriangledown l(w; b)$
    return w to server

2.1.1 FedPaq:

After the idea of federated learning was introduced many research has also been done how we can efficiently share weights between the server and the clients, thus decreasing the latency during the weight sharing and increasing the efficiency of the overall system. One such approach is proposed by the authors of [4].
In their proposed system they proposed three basic things which would help to improve the efficiency of such a system

1. Periodic Averaging : They proposed that rather than each client sharing its weights with the server just after one epoch, it should run τ number of local epochs on the local data before and then share the updated weights with the server. They also found out that if τ (number of local epochs on local data) is increased the number of total federated iterations (K) required could be reduced.

2. Partial node participation: In the classic federated learning algorithm the server could still choose all of the clients, however in this research authors claims that we don't need to pick all of the nodes. Choosing nodes randomly also gets us to the desired accuracy.

3. Message Quantization : Uplink bandwidth on the mobile devices could become a bottle neck for the device to efficiently share weights/updates with the server. The author proposed that if we could quantize the weights before sending it to the server we could save alot of bandwidth and still get similar results. They proposed a quantizer which was similar to the [6]

Algorithm 2 shows the FedPaq algorithm proposed by the authors. Here K is the K number of iterations given to federated learning, τ is the number of epochs for the local updates, S_k denotes the selected subset of k edge devices chosen by server for one iteration and Q is the quantization function.

Algorithm 2 FedPaq

for k = 0,1, .., K-1 **do**
  server picks r nodes S_k uniformly at random
  server sends x_k to nodes in S_k
  for node i $\in S_k$ **do** $x^{(i)}_{k,0} \leftarrow x_k$
   for $t = 0, 1, ..., \tau - 1$ **do**
    compute stochastic gradient
    $\nabla f_i(x) = \nabla l(x, e) for\ a\ e \in P$
    set $x_{k,t+1} \leftarrow x_{k,t} - n_{k,t} \nabla f_i(x_{k,t})$
   end for
   Send $Q(x_k, \tau - x_k)$ *to the server*
  end for
  Server finds the weights and completes the update.
end for=0

From Algorithm2 we can see that, the server first selects r number of random nodes which will participate in the current learning cycle. Then those r nodes run τ number of local iteration on the data. Once the local training is completed, they then Quantize the weights/gradients based on the quantizer Q chosen. They can use a simple quantizer suggested by the authors in [6]. Once the weights are quantized they are shared with the server and server performs the FedAvg algorithm on the received weights and update the model.
In their experiments, authors claimed that by increasing τ the number of iterations required overall K decreases. Which implies if we give ample time for the neural

networks to train before averaging the weights, we can efficiently converge our overall model by running less number of iterations.

2.2 Adversarial Attacks on neural network:

Ever since the development of AI technology, a continuous and parallel research has also been going on about the adversarial attacks on such technology. The focus of such research is to show how anyone can steal the data from a trained AI model i.e to show the vulnerability of the current AI models so that they can be improved. Such attacks which try to spoof the trained AI or are aimed to extract information from such models are termed as adversarial attacks. This area of research is useful as it allows us to find the loopholes in our current architecture and build better models over them.
Soon after federated learning was introduced, researchers like [7][3][2] found some loopholes in the approach and claimed if the weights/gradient shared by the device in federated learning approach are not secured and if the security of the connection is compromised then the hacker can steal all of the data related to the machine using the shared weights. Such attacks aims to "find images that lead to a similar change in the model prediction as the ground truth". When the models share weights with the cloud server they can steal the data with 99% correctness if that device is malicious from the shared weights.

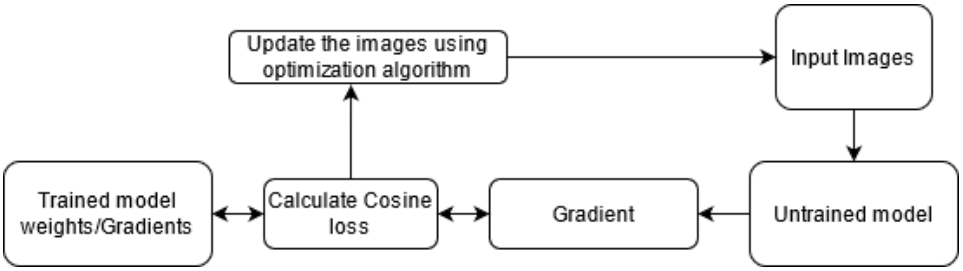

Figure 2.2: Flow showing how an adverserial attack can be trained to fool a neural network

The figure (2.2) illustrates what the general proposed attacking algorithm looks like. The basic idea is to use a similarity based cost function i.e cosine similarity along with some optimizer (eg: adam) to reconstruct the images using the shared gradients.

The attacking device start off with a random image, and then pass it through the untrained network and store the gradients for that forward pass. Since, the device has the gradient from a trained network, it can calculate the similarity loss such as cosine-distance between each of the gradients of each layer w.r.t the trained gradient. Once done it then updates the input image using a simple back propagation. Once the gradients of layer are similar enough the reconstructed image also become similar to the image user used to train its neural network and thus revealing the private information of the user. Using this approach authors have proved that they were able to extract information from different networks such as Res-Net and Le-net.

$$costFunction = \underset{x}{argmin} \mid \nabla L_\theta(x, y) - \nabla L_\theta(x*, y) \mid \tag{2.1}$$

Equation (2.1) shows the general cost function that the malicious server maximizes once it receives the gradients from the device. Here $\nabla L_\theta(x,y)$ represent the gradient send to the server by the edge device which it has calculated using its own data x,y and $\nabla L_\theta(x^*,y)$ represent the gradient of the server using the random set of data (x^* and y).

From the equation we can see that the server receives the update $\nabla L_\theta(x,y)$ from the edge device and it tries to close the difference between its own gradient $\nabla L_\theta(x^*,y)$ and the gradient received by the device by maximizing the cost function. and when it converges it can simply backpropogate those gradient to the images to find the images related to that gradient, thus breaching the privacy of the user.

2.2.1 How easy is it to break privacy in federated learning?

In [2] authors showed how easy its to break the privacy in federated learning if no data encryption is provided. Their experiments were able to reconstruct the source images from the gradients shared by the devices to the server.

Figure 2.3: Reconstruction of input image x from the gradient shared to the server. Left: Image from the validation dataset. Middle: Reconstructed image from a ResNet-18 Architecture. Right: reconstruction from a trained ResNet-152

Authors proposes that if we have a data x $\in \mathbb{R}$ then it can be recovered from its gradient G $\in \mathbb{R}$.
Lets consider that our neural network has a ReLU non linear activation function then the output of the layer would be $x_{l+1} = \max(A_l x_l, 0)$, where A_l can be considered as the weight matrix of the layer l and x_l is the input of the layer l. Also assuming that derivative of loss w.r.t the layer's output contains at least one non zero element then we have: $\frac{\partial L}{\partial x_{l+1i}} ! = 0$, here L is the loss of the network, then by chain rule we can extract x_l as

$$x_l = (\frac{\partial L}{\partial x_{(l+1)i}}) * ((\frac{\partial L}{\partial A_{li,:}})^T) \tag{2.2}$$

If we use (2.2) for each layer we can also use it for the very first layer whose input is the user data. Which would give us the input our user used to train the network before sharing its gradient, thus exposing its private information. Further more authors talks about that only the gradient magnitude captures the information about the state of training such as measuring local optimality of the data-point w.r.t current model. But the high dimensional direction of the gradient can carry significant information, as the angle between the two data-points quantifies the change in position at one data-point when taking a step towards other. Hence due to which authors of the paper suggested to use a cosine similarity function as it would capture the angle between two data points x and x*.

2.3 How about encrypting the input images:

Now that we know that it is easy to break the privacy of the neural network and we can extract information about the input data that was used to train the network from the network itself. Recent researches like [2] also suggests that we can extract such information even in a federated learning setting.
Since in a federated learning setting only gradients/weights are shared the attacker is able to extract the information from the shared gradients. However, if we train our neural network on encrypted data only, then an attack on such a setting will only reveal the encrypted information as that is the information that our network would have seen and the user private information will be preserved.

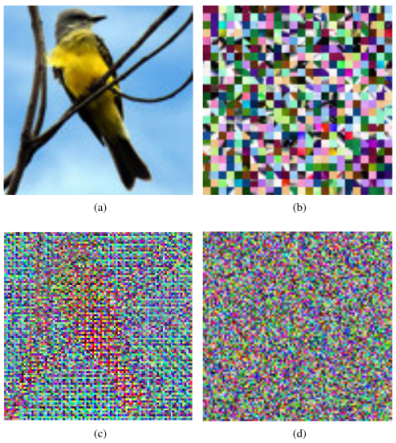

Figure 2.4: Images showing how image encryption can save user private information. a) shows the orignal image and b),c),d) shows the encrypted image of the same orignal image

The authors of [8] suggested an image processing technique to encrypt the image. Figure (2.4) shows the example of such technique. They suggested to use a negative positive transformation technique that we could apply at each pixel of the input image and which would change the input image (2.4) a) into the encrypted image (2.4) d).

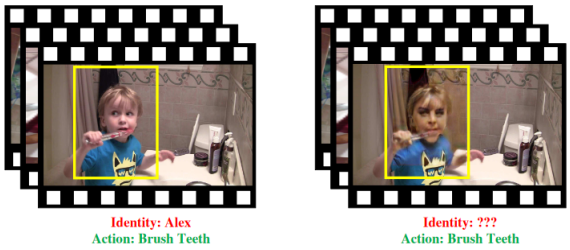

Figure 2.5: Images showing how GANs can be used to hide the sensitive information from the images.

Also there were other techniques put forwarded by the authors [9] where the authors would deliberately hide the private information of the user from the data using GANs and then we could use these new images to train and test our neural network. As shown in figure (2.5) the image from the left shows the orignal image with the user face in it which can be considered as the private user information, but in the image on the right we can see how the face is morphed with the face of a doll so that the user private information is still preserved.
A positive side of using such techniques is that since we are using encrypted images to train our neural network, if the attacker has the access of our network weights then the weights will only reveal the encrypted information i.e information which is unrecognizable by humans. However in all of these approaches we needed to retrain our neural network with the new augmented data. But in federated learning setting we have a global model which we train on a standard dataset for hundreds of hours to get State of the art accuracy which is then shared among all the edge devices. Training the network again would be a huge waste of the computing power. So we need an encryption technique which we could use without retraining our neural network again.

2.4 Keynet- A secure way of encrypting weights:

While AI models were beating humans in terms of accuracy in most of the domains such as face recognition there was also a rising concern about the security of such models. As the accuracy of such models have increased over the years, so has the number of adversarial attacks one can do on such models to extract private information. Although Federated learning was introduced to save user private user information al-

gorithms studies such as [3][2] suggests that there are ways in which a hacker can still get the user private information by using the shared weights with the server.
To tackle such attacks researchers have developed numerous algorithms [10][8][9] where they can preserve privacy of the user by having an additional encryption step. Either researchers would train an autoencoder which would hide the user sensitive information from the data before using it to any other task [9], or they would simply change the values in the data based on a predetermined algorithm. By doing so the data would be uninterpable by humans but we can still train a neural network and get good results out of it. However in most of the research we either need to retrain the model to use the new input as we can not use our current SOTA model which we trained for hours and in some cases as using an autoenconder to hide user information the preprocessing step would consume a lot of time and resources thus increasing the inference time of our neural network.
These challenges were solved by the authors of [5]. They built an efficient Homomorphic encryption system for convolutional neural networks which do not require any retraining of our SOTA neural network or an additional preprocessing step.
Convolution is a linear operation and a convolution neural network N(x) can be thought of as the Network consisting of N_k such linear-layer-wise function.

$$N(x) = N_k(N_{k\text{-}1}(...N_1(x))) \tag{2.3}$$

We also know that since it is a linear operation we can represent it as a sparse-toeplitz matrix where the kernel is replicated row wise. The dense multi-channel input vector can also be represented as a flattened array and then the output of the layer can be calculated using the matrix multiplication of the weight and the flattened array. Keynet builds on this idea of linearizability of the convolution neural network. They use private layer keys A_i to transform the network weights W to $\hat{W}$ using

$$\hat{W} = AWA^{-1} \tag{2.4}$$

such that the source weights cannot be factored to recover either A or W. Here W are the layer weights and A are the respective layer keys.
They also define certain conditions which for a function to be a optical tranformation

function.

1. Linear: The function f to be linear (f==A).
2. Invertible: The matrix A must be positive definite.
3. Non-negative: A $\geq$ 0 for all matrix elements.
4. Commutative: There exists a non-linear activation function g that is commutative with A, such that
$$A(g(A^1x)) = g(AA^1x) = g(x)$$
5. Sparse : The private keys A should be sparse.

According to the authors since optical image formation can be modeled as a linear transformation, condition 1 is used to make sure that transformation functions are also linear. Condition 2 is used to make sure that the original image can be recovered by $A^{-1}Ax$. Condition 3 limits a matrix to be physically realizable. Condition 4 enables inference with networks having non linear activation. Condition 5 ensures that end-to-end inference in the optically encrypted convolutional network is efficient and does not require an infeasible dense matrix.
Now if we have a secret layer keys A_i and a secret image key A_0 then our network can become:

$$N(x; W) = A_k W_k ... (A_2 W_2 A^{-1}{}_1)(A_1 W_1 A^{-1}{}_0) A_0 x \tag{2.5}$$

where x is the input image and W are the weights of the convolutional layer.
But by condition 4 we can only use a non linear activation function g such that g and A are commutative:

$$Ag(A^{-1}\hat{x}) = g(AA^{-1}\hat{x}) = g(\hat{x}) \tag{2.6}$$

For the paper they showcased the transformation using a Generalised doubly stochastic matrices as it fulfilled all of the conditions for their transformation function and the choice of activation function was chosen to be g=ReLU.
A doubly stochastic matrix is a non negative matrix such that each row and column sums to one and it is well known that every doubly stochastic matrix can be decomposed into a convex combination of permutation matrices. A permutation matrix $\prod$

is a square matrix that has exactly one entry of one in each row and each column and zero else where. A Generalized doubly stochastic matrix has arbitrary non-zero entries without requiring the rows and columns to sum to one. Such matrices can also be defined as the product of a diagonal matrix D and as a convex combination of α permutation matrices $\prod(i)$. Equation (2.4) shows how we can create a generalized doubly stochastic matrices using permutation matrices and a diagonal matrix.

$$P = D \sum_{i<=\alpha} \theta(i) \prod(i) \tag{2.7}$$

Here $\prod = \sum \theta(i) \prod(i)$, such that $\sum_{i<=\alpha} \theta(i) =$ and $\theta >= 0$. θ is a hyper parameter that is used to ensure that our permutation matrix $\prod$ is positive and definite. The parameter α is used to ensure the softness of the matrix, i.e if $\alpha = 1$ then our stochastic matrix becomes a permutation matrix.
First they use a private image key A_0 to encrypt the image and then they send the encrypted image to the encrypted neural network and get the output. Now using such an approach we would not have to re-train our neural network on the new transformed image as this approach can be used to encrypt the network and the image and give the correct output.

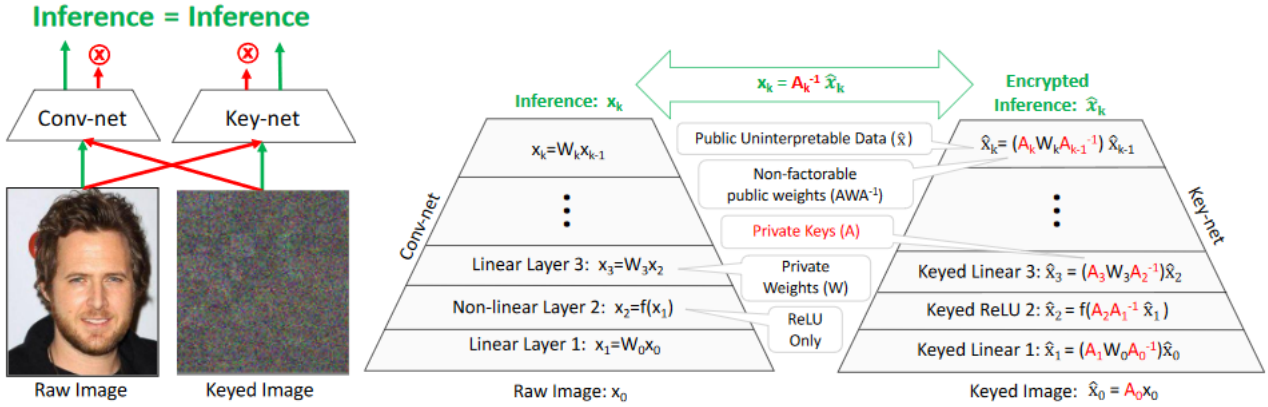

Figure 2.6: Image showing how the encryption in the keynet works

From (2.6) (left) we can see that the image on which inference is performed on keynet is not interperible by the humans, and also if the original image is fed into the keynet or vice versa then the output produced by the network will be wrong and the

attacker wont get any information out of it. From the image on the right we can see how a keynet encrypt the image/network pair. First using a stochastic matrix the input image is encrypted, and then that encrypted image is fed into the key net network and produce the desire output.

2.5 MobileNet

Convolution neural networks[11] are widely used for image recognition tasks. These networks consist of a convolution layer followed by an optional fully connected layer. Each of the convolution layers takes the image as an input and produces an output which represent some information it has learned from the image.

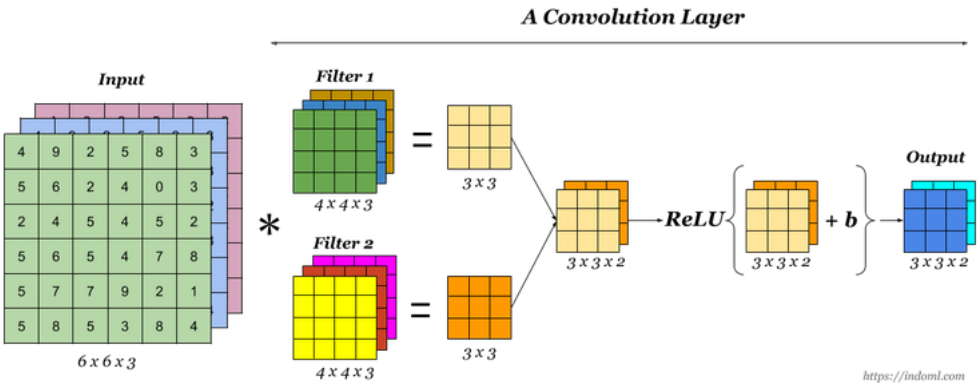

Figure 2.7: Image showing how convolution operation works

From the figure above we can see how a convolution layer computes its output when a 2d image input is provided to it. In a typical convolution layer input is first convoluted with the kernels of the layer size k_{h}(kernel height) and $\mathrm{k_w}$ (kernel width). Each kernel has channel equal to the channel of its input, in the figure we can see that each kernel has the channel size of 3 which is equal to the channel of its input. The number of kernels are determined by the number of output channel the convolution layer has, in the figure number of output channels are two thus we are having two kernels. Each of the kernels are individually convoluted with the image and there output are stacked together and then is passed to a Non-linear activation function (ReLU)[12] in the image and then the output is passed to the next layer in the network.

Lets calculate the number of multiplications required by our convolution layer to provide the output. Let's say that our kernel size is k_h * k_w * k_n where Kh is the height of the kernel, Kw is the width of the kernel and kn is the number of kernels. Let's say we are producing an output of having channel size D_o then the number of operations becomes

$$Numberofoperations = D_o * k_h * k_w * k_n \tag{2.8}$$

and if we have n such kernels our number of multiplications becomes

$$Numberofoperations = D_o * k_h * k_w * k_n * n \tag{2.9}$$

There are multiple such convolutions in our network and each having its different dimensions, hence it might get difficult to run our traditional network on the mobile devices. To mitigate this, authors from [13] came up with the idea of mobile net. Here the traditional convolution layer was replaced by a depthwise separable convolution. A depthwise separable convolution consists of a depthwise convolution and a point wise convolutions. By doing so the number of operations required in a single convolutiona layer decreased and the network also became fast by decreasing the number of operations but with an accuracy drop of just ¬2%. Such networks which use depthwise separable convolutions are called mobile networks[13].

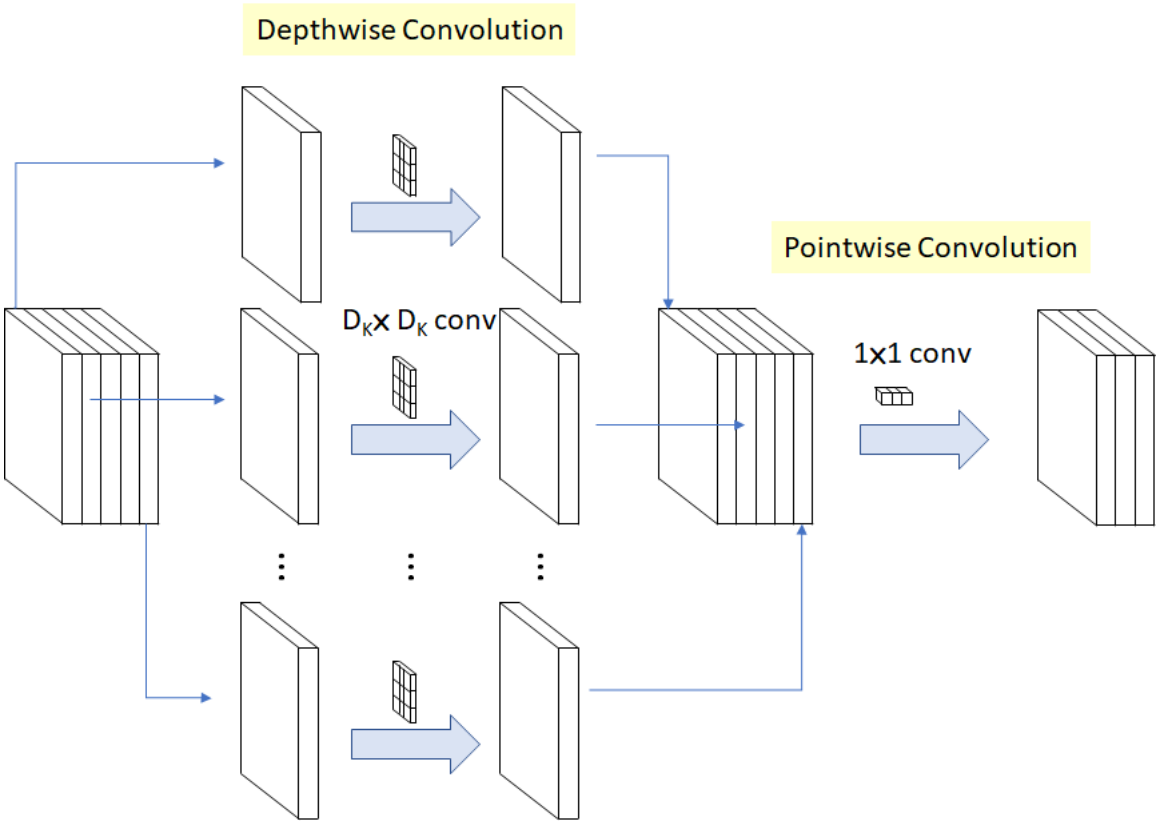

Figure 2.8: Depthwise separable convolution

Each Depthwise Separable Convolution layer consists of two operations namely Depthwise convolution and point wise convolution. The idea behind such layers is to have n (number of input channels) kernals of $k_h * k_w$ with channel one and then having an another convolution layer of kernal size 1*1*M, where M is the number of output channels, instead of having M kernals a size of $k_h * k_w * k_n$. This helps in decreasing the number of calculations required per layer, and thus makes our network smaller and faster.

2.5.1 Depthwise Convolution:

Lets say we have an input with M number of channels then our depthwise convolution will have M kernals with a size of $D_k * D_k$ as shown in the figure. So the number of operations required to perform the depthwise convolution on an image would be

$Df^2 * M * DK^2$

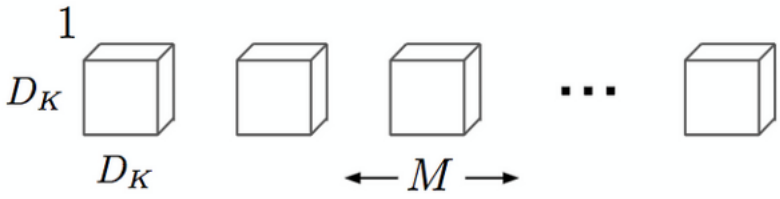

Figure 2.9: Depthwise convolution

2.5.2 Pointwise Convolution:

In a point wise convolutions we have a kernal of size 1*1 each having M chanels and we have N such kernals, where N is the number of output channels. So the total cost of point wise computation becomes $Df^2 * M * N$

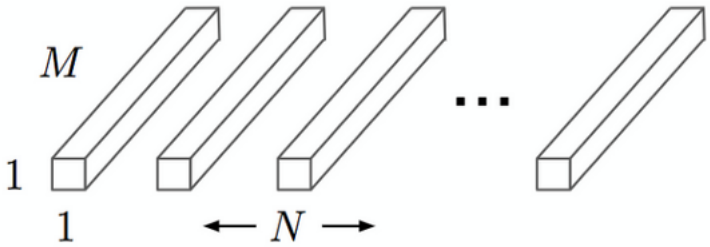

Figure 2.10: Depthwise convolution

So the total computation of a depthwise separable convolutioal layer becomes C = Cost of depthwise convolution + cost of point wise convolution which becomes :

$$C = Df^2 * M * DK^2 + Df^2 * M * N \tag{2.10}$$

From (2.9) (2.10) it is clear that the number of operations in a depthwise separable convolution is less than a normal convolution layer. Thus this makes mobile net much faster and an optimal choice to be used for edge devices. Hence for this dissertation we will be using the mobile net to demonstrate the results we acquired by running multiple experiments.

Chapter 3

Design and Implementation

3.1 Overview

By our experiments we are aiming to provide results that using the [5] and federated learning we can improve the security of the federated learning setup without a drop in accuracy. In this chapter we will take an in depth look about the over all architecture of our system and how we implemented different things that were needed for the project.

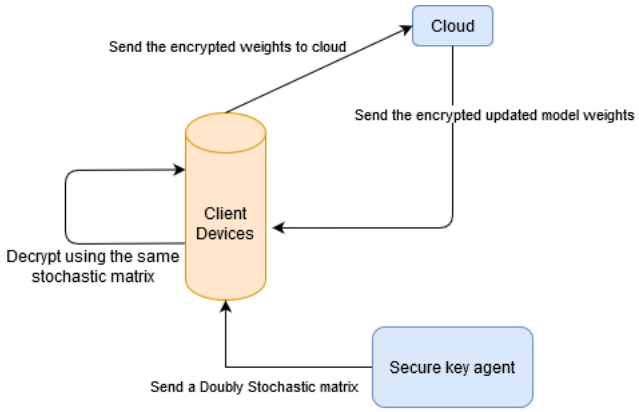

Figure 3.1: Proposed Architecture

The figure (3.1) shows the image of our approach. We have a secured key agent which each of our client will communicate with to get the the secure key. Once the key is received the clients will encrypt these weights using the key provided and will share the weights to the cloud. The cloud will do the updates on the encrypted weights and

then share the updated weights with the clients. Clients will then again communicate with the secure key agent to get access to the decryption key and use it to decrypt the received weights and update their model. Having such a approach is beneficial as even if the security of the cloud server is compromised it can not tell what the values of weights are as they will be encrypted.

Algorithm 3 provides a more in depth look at the process of how our framework is

Algorithm 3 Secure Federated Framework - Client Devices are the local edge devices with there private information. Secure key agent is the agent which the client communicate with to get the encryption and decryption keys. Cloud is the cloud server which performs the Federated averaging algorithm

Client downloads the global model from the cloud.
Each client trains the global model on their local data.
Each client chosen to communicate with the server, asks the secure key agent for the encryption key.
Secure keyagent provide the encryption key to all of the clients.
Each client encrypt the weights using the secure key.
Clients sends the encrypted weight to the server
Server performs the Federated averaging algorithm on the encrypted weights and shares back the updated global model to the clients.
Each client asks the Secure key agent for the decryption key.
Secure key agent provides the decryption key.
Each client decrypt the weights and start using the updated model.

used. Also keep in mind which clients are used in the current communication cycle and what federated averaging algorithm to use is entirely up to the user of the framework. As we later show that our framework is an abstract framework and can be used easily with any of the recent federated learning approaches.

3.2 Training Environment

The training for different models in a federated learning setting were performed on cloud with Google Colab Pro [14]. Google Colab is a free jupyter notebook based environment on frontend and linux on the backend. It is a product of Google to encourage Machine Learning projects and research on its platform. However, being free, there are lot of

constraints on the product, such as 'idle timeouts' of the notebooks, '12 hours run-time limit', with GPUs of Nvidia K80 and lower RAM.
Free version of google colab was working fine until the very later stages of our project. So we decided to use the local machine to do run the experiments. Our machine had a 6Gb Nvidia 1660TI Gpu which we used to train the adverserial attack for our algorithm. Along with the gpu our machine had 16Gb of ram and an 6cores i7 processor.
Our federated learning setup was entirely built on pytorch [15] and the ideas were taken from [16]. Our secure keyagent was based on the open source code provided by [5] and [17]. For our research purposes we emulated the federated learning environment with only two users and tried two federated learning approaches namely [1][4].

3.3 Dataset

We trained our model for a classification task as the models in such a setting are easy to train and there are many preprocessed dataset for the same.
For our experiments we used the CIFAR 10 dataset. We setup a google colab repo to conduct the experiments. To train the networks in a federated learning approach we used a non IID sampling of data i.e : both of the federated machine have different classes locally, and they get to see the data from other class once they received the updated weights from the server.

3.4 Network

Once our dataset and environment was decided. We needed to find what neural network architecture we can use to conduct our experiments. Upon further investigation we found out that mobilenet [13] is still a prevalent architecture which is used in the edge devices for the classification task. However such a network is still a big network and we would have to wait for hours to get results if we were just using it. So we decided to use Le-net as well. Le-net is a small convolutional neural network as compared to mobilenet and can provide the results in a matter of minutes unlike mobilenet which provided results in hours.
Moving forward we decided to first train and try to get the results on Le-net thus

guaranteeing the feasibility of our approach and then running the same experiments on our mobile net architecture as well.

3.4.1 Federated Learning Environment

For our experiments first we used the federated learning environment [16] which was based on the [1]. To run the basic tests. Once we got good accuracies on our base and encrypted model we updated the algorithm to incorporate the [4] and using some of the modules from [18]. After updating the environment we again ran the tests and tested how our algorithm performed under latest federated learning and the results are shown in the evaluation section.

3.4.2 Encryption and decryption of weights

We used the repository [5] provided by the authors of [5] to perform basic encryption of the weights. However they didn't provide any algorithm to decrypt the weights. So we updated the repository to incorporate the decryption mechanism for our framework.
We know that keynet perform the encryption of weights by multiplying the weights of layer l_i (W_i) of a network with doubly stochastic matrices A_i and ${A_{i-1}}^{-1}$. Where ${A_{i-1}}^{-1}$ is the inverse of doubly stochastic matrix A_i which is used to encrypt the previous layer. Since, all the network will have the same architecture and we will be providing the same private key to all of our networks which are participating in the current iteration, we can be sure that all the networks will have the same keys A_i for their layers.
So before sending the weights each of network encrypts their weights with the keys provided by the agent, and when they received their weights back from the server, they again asks the *secure agent server* for the decryption keys. Once these are provided we can simply perform a matrix multiplication as shown in equation (3.1) to get back the original updated weights.

$$W = {A_{i-1}}^{-1} \hat{W} A_i \tag{3.1}$$

Equation (3.1) shows how the original weights can be received from an encrypted weights if the current matrices are provided. Here $\hat{W}$ is the encrypted weights received from the server and A_i and ${A_{i-1}}^{-1}$ are the keys received by the secure agent server.

3.5 Checking for attacks

There are many papers such as [2] [3] which provide out of the box solution on how we can steal data from a federated learning setup. Once our model was giving good accuracy we decided to test our framework against a non encrypted federated learning setup using the attacks described by [2]. The purpose of these experiments was to show that our network not only provides the same accuracy as that of an unecrypted environment but also provides an extra layer of security of the user data.

3.6 Approach

For our experiments we trained the mobile net and Le-net in a federated learning environment. There were two federated users, each of the user had their separated classes which were present in the dataset. They would train locally and then would send their weights to a cloud server. The cloud would perform the averaging algorithm and would return the updated weights to the user along with updating the global model of the system.

We had two experiments running for each of the architecture i.e: mobilenet and lenet. In one they would perform the classic federated learning algorithm i.e approach without the encryption and in the other they would first encrypt the weights using the customized key-net library and then would send the weights, server would perform the averaging algorithm on the encrypted data and then send back the weights to the devices. The devices would get the updated weights from the server. They would then decrypt the weights and then start using the updated weights.

Once our models were trained we first compared the validation accuracy of both the models i.e one using keynet and without keynet. Once that was done we tried running the adversarial attack suggested by [2] on both types of model trained and checked whether we were able to extract information from the respective shared weights. Ideally the hacker should get an image closer to the original image if no encryption is used and should get just noise when our approach is applied to it.

After conducting the aforementioned experiments we tried testing different Federated algorithm like [4] to our approach and checked whether our architecture was easily applicable to new SOTA federated learning algorithms. Also applying [4] helps us see how

our framework would behave if we introduce some form of quanitzation while sharing the weights.

Chapter 4

Results and discussion

4.1 Evaluation

In this section we will showcase and discuss various experiments we performed during the course of the dissertation.

4.1.1 Training and evaluating the approaches

As discussed before we trained two network architecture LeNet and MobileNet on Mnist and CIFAR10 dataset respectively. Both of the networks were trained for a classification task. For each of the architecture, we first trained and evaluated the results in the basic federated learning setup proposed by [1] which we considered as our baseline model. Then we introduced the keynet encryption while sharing the weights and compare the evaluation performance. Once the model were converging and were giving almost similar accuracies we tested the both unencrypted and encrypted model against the adversarial attack suggested by [2].
Once these two experiments were confirmed we then tried running the FedPaq algorithm on our architecture with and without encryption and compared that results as well. The last experiment was done so as to make sure that our framework fit seamlessly with the new Federated learning algorithm that are being developed recently.

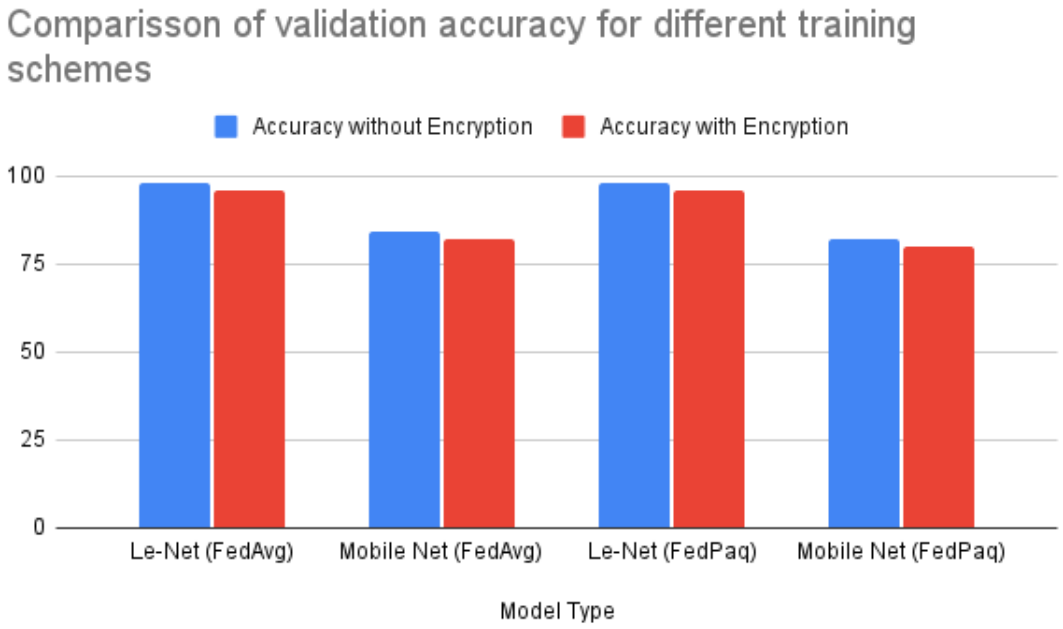

Figure 4.1: Chart showing validation accuracy comparison of various training schemes

Model Type	Accuracy without Encryption	Accuracy with Encryption	Federated Algorithm
Le-Net	98	96	FedAvg
Mobile Net	84	82	FedAvg
Le-Ne	98	96	FedPaq
Mobile Net	82	86	FedPaq

Table 4.1: Table showing validation accuracy comparison of various training schemes

From table 4.1 and figure (4.1) we can clearly see that models trained with our approach clearly converges, however there is an accuracy drop of almost 2%. This can happen because of how pytorch (framework we used for our mobile network) and numpy (a general purpose array library in python) handles the large floating point, due to which some of the floating point digits occurring at the end of the floating point number might get changed and thus we see an accuracy drop.
Another interesting thing to note here is that, while Le-net almost provide similar accuracies for both FedAvg and FedPaq (with or without encryption), there are some accuracy drops while running the experiments for the mobile net. We think this at-

tributed to the fact that since Le-net is a very shallow model as compared to mobilenet. Thus quantising the weights didnt have much effect on the network accuracy. But it can also be the case since mnist is a much simpler dataset as compared to the Cifar-10, as it only contains grayscale images of digits starting from 0-9, it is very easier for even a shallower network like Le-net to classify them easily. Thus even after introducing some quantisation there is no loss in the accuracy.

4.1.2 Testing our trained models against the adversarial attacks

Once our models were trained and we were getting decent accuracy (approx 80%)in our baseline models and trained encrypted models we first tried to check how the images fed to our encrypted and unencrypted netowork compared and whether we could infer anything about the data from those images. .

Figure 4.2: The image fed to our neural network for inference

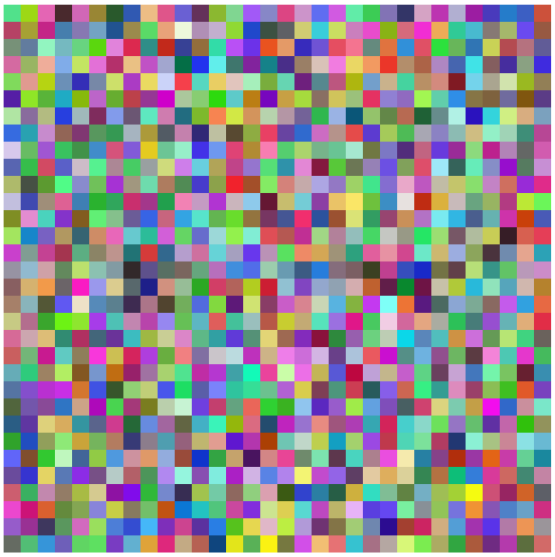

Figure 4.3: The image fed to our encrypted neural network for inference

Both the images (4.2)(4.3) represent the image which are fed to a neural network. Fig(4.2) is fed to an unencrypted network, and the fig (4.3) is fed to the encrypted network with the same architecture. For both the images the respective networks produced the same output i.e the class representing the image. However we can clearly infer what information was present in the unencrypted image (4.2) but it is hard to infer the information present in the image (4.3). This conforms to our hypothesis as the images send to our neural network are barely readable to human eyes thus if the hacker even exploit our neural network to get those images they will only get data that is close to just noise and wont be able to interpret anything from it.
After this we introduced a malicious server and tried replicating the attack suggested by the authors [2] as this would replicate the real world scenario where the hacker will try to steal the data from such network using such attacks. Once the weights were shared with the server after each epoch, the server would launch an attack similar to [2] and then try to find the host images. In the following section we will demonstrate what were the images our server was able to reconstruct after the attack.

Figure 4.4: Figure Showing the orignal CIfa10 images we used to reconstruct.

Figure 4.5: Figure Showing the Reconstructed images if the security of the server is compromised and the model is not encrypted.

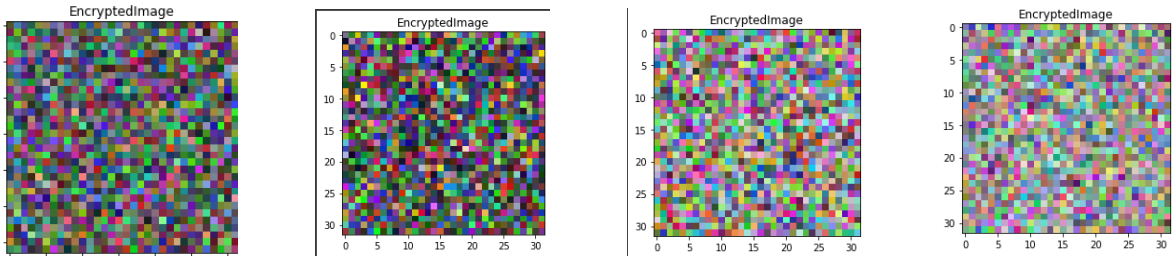

Figure 4.6: Figure Showing the the encrypted images when keynet module is used.

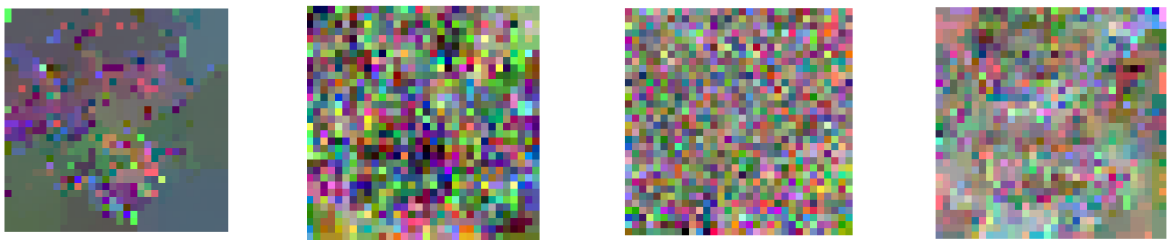

Figure 4.7: Figure Showing the reconstructed image, if the attacker tries to reconstruct images but the weights shared are secured.

Figure (4.4) were the images our network saw before sending the weights to the malicious server, however from figure (4.5) we can see that our malicious server was able to reconstruct the images and was able to see the information present in the images just by using the weights shared by the client. However if we shared the encrypted weights, i.e: encrypting the network with the keynet and then sharing the weights. Then figure from (4.6) we can see that the attacker was not able to reconstruct the original data but was only able to see random pixel. Which in turn confirms our hypothesis as if we encrypt our weights before sending it to the server it in turns increases the security of our system.

4.2 Discussion

Our framework tries to decouple the encryption part and the federated learning part. Thus making it hard for an attacker to steal data from the shared weights/gradients. To access information from the share weights in our framework, the hacker not only have to first access the layer keys from the secure key-agent, it then also had to access the corresponding the network keys. Which makes the tasks really hard and accessing user information a lot much harder.

From the accuracy comparison of our framework with the baseline model we can safely see that in our framework the neural network converge and give almost similar accuracy as that of the baseline models. Also we would like to point out that in our technique if an attacker tries to steal the data from shared weights, it will only be able to see the encrypted images and not the real images as evident from the images figure (4.3).

Also the overhead computation of encrypting the network in our system depends on a lot of factors such as layer dimensions, number of layers etc. At the backend we have used [17] to write subroutine to encrypt and decrypt each layer faster and efficiently. However as we try and incorporate more network in the future we might need to write more sophisticated subroutines to make sure that the over head computation cost for encrypting the network is negligible.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

After performing and conducting multiple experiments we can safely conclude that the framework provided by us can be used as a way to preserve user privacy in a federated learning setup.
Also an important aspect of our approach is we decouple the secure agent from the main server thus it makes the attacker hard to attack our system. As it first needs to find what key we are using from the secure key agent, then if it correctly found the key then again it had to intercept the weights during the correct instance, only if it is successful then it had to first decrypt the correct weights and then run the appropriate attacking algorithm.
We have tested our approach on latest neural network architecture which is used most for mobile edge devices named mobilenetv2 [13]. While conducting the convergence test we used the basic federated learning algorithm as our baseline model and compared our encrypted federated learning algorithm against the accuracies of the baseline as shown by figure (4.1).Apart from that we have tested our approach on various federated learning tech such as [4] and [1] thus making sure our approach can be easily integrated with various federated learning approaches. A negligible accuracy drop can be found between a baseline and the encrypted model and we assume that happens when we encrypt our weights and can be attributed to the fact how numpy and pytorch handles the floating point numeric in the back-end.

Also we used attacks on the server suggested by [2], thus trying to simulate the environment what would happen "if a hacker have access to our server and tried to steal the data", the results shown from (4.6) proves that even if the hacker have access to our global model they will only be able to see just noise and not the user private information from that model as we are sharing the encrypted weights not the usual weights thus confirming that we can use our framework to protect private information of the user in a federated learning environment.
The overhead computation for encrypting the weights is depended on the number of layers and how many parameters each layer had. In our case for both mobile-net and Lenet overall computation cost to do the encryption was negligible. However for different network architectures if the computation costs increases more optimal subroutines in numba [17] can be written to optimize the cost.

5.2 Future Work

As shown in the evaluation, our approach as of now can be used with most of the federated learning frameworks available. However there are numerous ways in which we can improve such a system. Right now we are only using single private key which is shared to all of the users used to encrypt the weights. Moving forward we can develop a system where we can use different keys for different users to encrypt their weights, thus increasing the encryption technique provided by our system. Also in that scenario a more sophisticated key management algorithm should be built and we should again test the convergence of such an algorithm.
Apart from this there are many different ways in which we can encrypt a model weights using the keynet library in this dissertation we have only experimented with a few, moving forward we can look at other techniques as well and choose the one which suits best for our need.
Also right now keynet [5] only supports ReLU[12] activation only. This limitation is due to the fact they couply encrypt the neural network and the image together. Moving forward we can also look at ways on how we can overcome this situation so that all the other activation functions can still be included.
Apart from that keynet also have multiple constraints on the parameters of the network layers, for eg : it only supports the convolution neural networks [11] as of now and it

has constraint on what pooling layer to use. Moving forward we can look for different ways in which we can over come these short comings, as doing so it will make our framework acceptable to other types of neural networks such as LSTMs thus we can use it for more than just image recognition tasks.

Bibliography

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

[2] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients – how easy is it to break privacy in federated learning?," 2020.

[3] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Advances in Neural Information Processing Systems*, 2019.

[4] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani, "Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization," in *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics* (S. Chiappa and R. Calandra, eds.), vol. 108 of *Proceedings of Machine Learning Research*, pp. 2021–2031, PMLR, 26–28 Aug 2020.

[5] J. Byrne, B. DeCann, and S. Bloom, "Key-nets: Optical transformation convolutional networks for privacy preserving vision sensors," *CoRR*, vol. abs/2008.04469, 2020.

[6] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[7] T. Orekondy, S. J. Oh, B. Schiele, and M. Fritz, "Understanding and controlling user linkability in decentralized learning," *CoRR*, vol. abs/1805.05838, 2018.

[8] W. Sirichotedumrong, Y. Kinoshita, and H. Kiya, "Pixel-based image encryption without key management for privacy-preserving deep neural networks," *IEEE Access*, vol. 7, pp. 177844–177855, 2019.

[9] Z. Ren, Y. J. Lee, and M. S. Ryoo, "Learning to anonymize faces for privacy preserving action detection," 2018.

[10] Z. Ren, Y. J. Lee, and M. S. Ryoo, "Learning to anonymize faces for privacy preserving action detection," 2018.

[11] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, *Object Recognition with Gradient-Based Learning*, pp. 319–345. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.

[12] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, JMLR Workshop and Conference Proceedings, 11–13 Apr 2011.

[13] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018.

[14] . Google Inc., "Google colab pro." "`https://colab.research.google.com/signup`". [online] Accessed: 31-Aug-2020.

[15] . Facebook LLC., "Pytorch." "`https://pytorch.org/docs/stable/index.html`". [online] Accessed: 1-Aug-2021.

[16] . Open source, "fedavg." "`https://github.com/AshwinRJ/Federated-Learning-PyTorch.git/`". [online] Accessed: 1-Aug-2021.

[17] . Open source, "numba." "`https://numba.pydata.org/`". [online] Accessed: 1-Aug-2021.

[18] . Open source, "Quantization in distributed learning." "`https://github.com/xinyandai/gradient-quantization.git`". [online] Accessed: 1-Aug-2021.

Appendix A

A.1 Converting Depthwise seperable convolution to a normal convolution

Depthwise seperable convolutions really makes a network faster, but most of the libraries usually do not provide out of the box support for such layers eg: keynet. In this section we are briefly going to talk about how we can convert such depthwise seperable convolutions into normal convolution, so that we can use them.

A.1.1 Depthwise seperable convolutions in pytorch

Since we are using pytorch in our implementation, we are going to first talk about how these layers are implemented in pytorch and then how we can convert them into a normal convolutions. As discussed before, Depthwise seperable convolutions consist of depth wise convolution and point wise convlutions. To implement depth wise convolutions in pytorch we can use the *nn.conv2d* module under the torch package. Such a layer can be intitalised as

$$\begin{aligned} layerDepthwise = nn.conv2d(inputChannel = input, outputChannel = input \\ groups = input, width = kw, height = kh, padding = kp) \end{aligned} \tag{A.1}$$

(A.1.1) represent how we can intialise a depthwise convolution layer in pytorch. The trick here is to set groups = number of input channels (input in our case) and setting the outputchannel = input channel (input in our example). Once we intialised

the depthwise convlution we can then initialise pointwise convolutions as

$$layerDepthwise = nn.conv2d(inputChannel = input, outputChannel = output \\ , width = 1, height = 1, stride = 0) \tag{A.2}$$

Equation (A.1.1) shows how we can initalise a point wise convlution layer in pytorch. The important thing to notice here is that, this layer looks exactly like the normal convolution layer, but it has a kernel of width and height equal to 1 and stride = 0. More information about such layer can be found at [15].

A.1.2 Understanding the Groups parameter.

From equation (A.1.1) and (A.1.1) we can see that point wise convolutions have similar definitions as that of a normal convolutions layer. However depthwise convolution is a special case of convolution as it had a group parameters and have output channels = input channel. So to convert a depthwise separable convolution layer we just need to convert depthwise convolution into a normal convolution and the rest of the network can just work as it is.

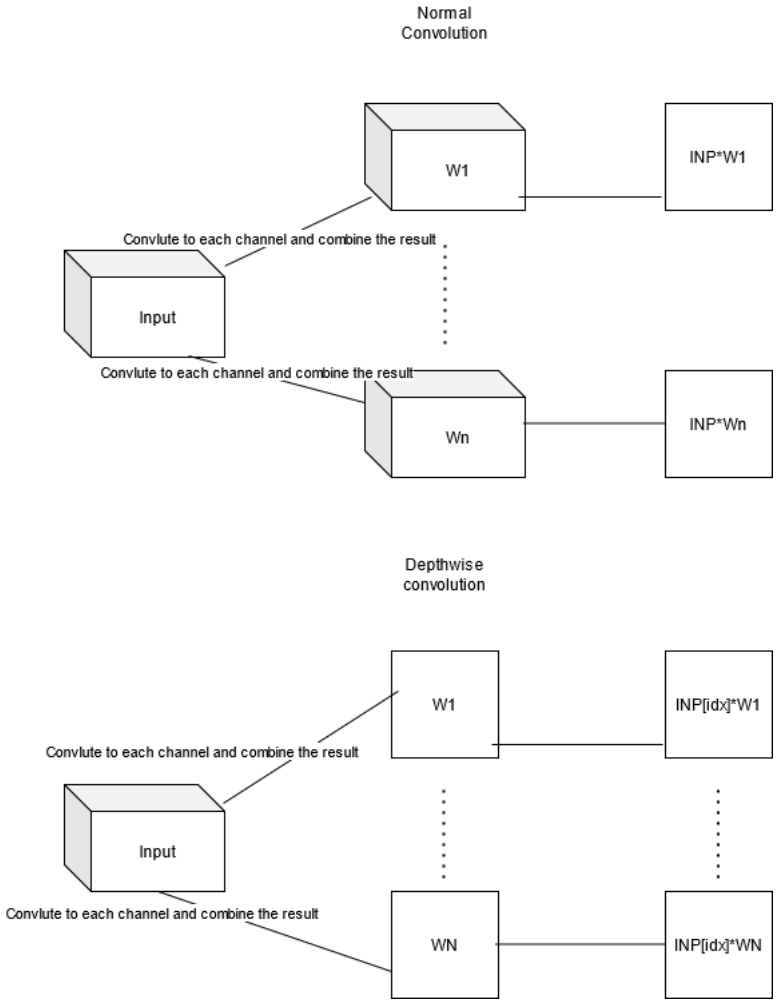

Figure A.1: The Figure shows how a normal convolution and a convlution with *groups* = *input* works when defined in pytorch.

From (A.1) we can see that in a normal convolution weights also have a depth attribute which is equal to number of channels in the input. During the convolution operation each of the weight convolute separately with the whole input and produces an output. Which is later stacked together to produce the output of the layer as a whole. Note: the number of weights here corresponds to the number of output channel. However if we give groups=input channel, then each of weights have depth = 0, i.e the are only square instead of a cuboid, and number of weights = number of input channels as shown in the figure. Each of the weight convolutes with their respective

input channels and produces the output. Output then again are stacked and are sent to the next layer. One quick fix to convert such a layer into a convlution layer can be to add the depth parameters to the weights in such convolutions by padding them with zeros. Note the depth added must only be equal to the number of input channels. However we must ensure to pad zeros in such a way so that the weight in the i^{th} position will have non zero values at the i^{th} position in the array.