Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Robust Federated Learning Approach in Heterogeneous Environment

by Ujjawal Aggarwal

Final year project
in completion of the BAI Computer Engineering
School of Computer Science & Statistics
Trinity College Dublin, Ireland

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university, and it is entirely my own work.
I agree to deposit this thesis in the Universitys open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgment.

Ujjawal Aggarwal

May 19, 2021

Acknowledgments

I want to express my thanks and gratitude for the precious contribution to my supervisors, Prof. Aljosa Smolic and Dr. Cagri Ozcinar. They helped me in the development of this project from scratch to the end. They both guided me throughout the project, mentored me very patiently, and gave me very insightful feedback, which significantly improved this work. Without their continuous support, advice, and assistance, this project would not have been not possible.

I acknowledge my sincere indebtedness and gratitude to my parents. I am thankful for their sacrifice, patience, and understanding, without which this would not have been possible. Its their devotion, support, and faith in my ability that helped me achieve my dreams.

Abstract

Recently, a new technique has been considered for training large machine learning models without gathering and storing user data, known as Federated Learning (FL). It allows companies to train large models collaboratively using user-end devices like laptops and mobile phones and allowing users not to share their sensitive information with third parties.

But Federated learning has its shortcomings when it applies to real-world scenarios like network optimization, low processing power on users end devices, etc. One of the critical challenges is Statistical heterogeneity, and it can be defined as "when devices frequently generate and collect data in a non-identically distributed (Non-IID) manner across the network".

In this paper, we proposed a robust approach by using Image Augmentation techniques in federated learning, which helps to overcome the problem of Statistical Heterogeneity. We will be using CIFAR10 and MNIST image datasets and applying Image Augmenting techniques like Cropping, Flipping, etc., to cure the Non-IIDness of the dataset.

The elemental part of this method is to increase the samples for the class which has fewer images(minority class) using image augmenting techniques without sharing the information amongst the users. We will add 5-6 unique augmented versions of each minority class sample image to the user's dataset to tackle the non-iid distribution. Our method will increase the accuracy by 4% for any p value(the portion of the dataset that handles the non-iid distribution) and 1.7% for any number of users, taking Google's FedAvg Model as the baseline.

Contents

List of Figures

Chapter 1

Introduction

This chapter details the background of machine learning, significant challenges related to machine learning, and then discuss new techniques like Federated learning, which overcomes storing user-sensitive information. Then finally discussing the shortcomings of this Federated Learning approach.

1.1 Traditional Machine Learning Approach

There is no doubt that Machine Learning plays an essential role in every industry, whether it is healthcare or finance or e-commerce or automobile. It has varied use cases like the COVID test using voice or forecast stock prices or product recommendations. Predicting the power of Machine Learning helps industries to take their next steps and boost their revenue strategically. But one of the major challenges in this state-of-the-art technology is gathering, constructing, and storing the dataset, which can be proved very difficult [2].

Also, the user needs to share their information that is relevant to the use case. As in product recommendation, the user needs to share their past order history, search history and clicks on the page, demographic info, gender, age, etc. All this information helps companies to create the best product recommendation algorithm for their customers, and it helps them to directly select from the recommended products instead of spending a lot of time on the website. Models learn and capture more underlying relationships within the data with sensitive information than non-sensitive information. Although, all the information that users share will be stored on the company's server in a very secure way.

But recent data breaches and leaks from big tech companies like Adobe [3] and Facebook [4] results in the misuse of user-sensitive information, and hackers sell this information for just some amount of money. To prevent this from happening, there are many strict guidelines and policies that stop companies from storing sensitive data. All this makes users think more before sharing their data with third

parties.

1.2 Federated Learning

Recently, a new approach has been considered for training large machine learning models without gathering and storing information centrally known as Federated Learning [5]. It is a distributed machine learning approach that enables companies to train on a large corpus of decentralized data residing on devices like mobile phones and laptops. In simpler words, federated learning can be defined as 'bringing code to the data instead of data to the code' and solves the fundamental problems of privacy and data ownership.

Large tech companies like Google [6] and major pharmaceutical companies like Amgen, Merck [7] use this Federated Learning approach to make their businesses more user-friendly, keeping the privacy concerns intact.

Potential applications of federated learning include a lot of tasks like learning from mobile users, predicting different health issues like heart attack from smart wearable, analyzes credit score, and knows how to prevent fraudulent activities from user activities. The two canonical applications in more details:

- **Learning over mobile devices**: Federated learning can help in predicting the next word by learning from users writing style. Users don't have to share their text data like messages, emails with third parties. Other substantial use cases include face detection and Voice Recognition. It offers excellent predictive powers on smartphones without diminishing the user experience or leaking private or sensitive information.
- **Learning across organizations**: Federated Learning can be very useful in the healthcare sector. It can predict different major health issues like breast cancer from X-ray images or COVID tests using the patient's voice. Hospitals operate under a stringent privacy policy and may face legal and administrative ethical constraints to keep patient data locally. Federated learning can help in this case, as hospitals don't need to share the database and still can use state-of-the-art technology.

Federated Learning offers a lot of prominent benefits to its users. Some of them are as follows:

- **Ensuring privacy**, as the sensitive user information resides on their devices only. Also able to benefit from training on a large joint dataset.
- **Smarter Models**, because of the collaborative training.

- **Lower Power Consumption**, as the training process on the user devices instead of a centralized server. It helps to saves a lot of energy and generate lower heat.

1.2.1 Difference between Distributed Learning and Federated Learning

Distributed machine learning refers to multi-node machine learning algorithms and systems that are designed to improve performance, increase accuracy, and scale to larger input data sizes [8]. It allows companies to make informed decisions from a large amount of data. Large and growing input data size helps many complex algorithms to reduce learning error and increase accuracy.
Distributed Machine Learning environment mainly involves conceptually converting the single thread algorithms to parallel algorithms. Also, large centralized datasets are replaced by high volume, high-velocity data streams generated by vast numbers of geographically distributed devices like sensors, autonomous vehicles, mobile phones, etc. So, the data acquired in this approach is in the form of IID or heterogeneous. But, the disadvantage of this approach is the requirement of sharing sensitive information and high power consumed by the large server.
On the other hand, Federated Learning is a particular case of distributed machine learning. The main difference between federated learning and distributed learning lies in assumptions made to its dataset as federated learning aims to train on the non-heterogeneous. In contrast, distributed learning originally aims at parallelism computer power. Distributed learning also aims at training a single model on multiple servers with the assumption that datasets are identically distributed, but none of the premises can be made for federated learning.

1.2.2 How Federated Learning Works

This section details the working of federated learning, which is not very complicated. The fig1.1 explains the mechanism of federated learning in a very simplistic manner.

1. Initially, the company or researcher will train the basic model on a small set of datasets and deploys it on millions of devices, as shown in fig1.1.

2. Those users will use this initial model on their devices where the sensitive information is stored. The model will learn from their data using the processing power of their devices, as depicted in step A of fig1.1.

3. As there will be millions of learned models, all of them will be automatically upload on the companies cloud server as shown in step B of fig1.1.

4. Afterwards, all the model weights will be combined into the single weight file using the different averaging algorithm as conveyed in step C of fig1.1.

5. Then again, this newly developed model will be deployed to its users for their use and model learning. This cycle goes and goes till the company achieves some good results.

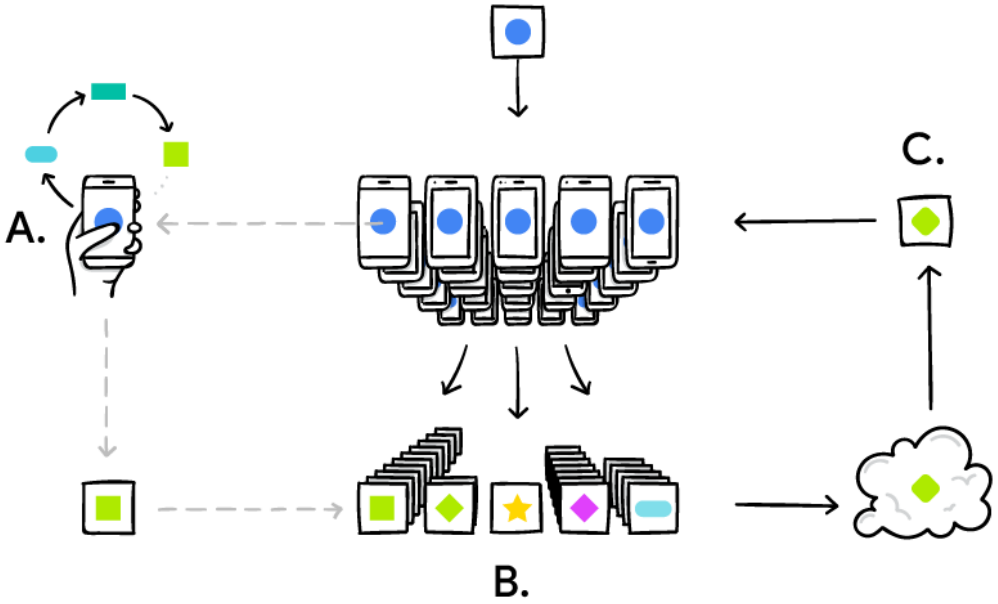


Figure 1.1: Working of federated Learning [1]

1.2.3 Challenges in Federated Learning

Federated Learning has shown a great deal of improvement on both the Accuracy and Loss curve. But FL has its own shortcomings when it applies to real-world problems [9]. Some of the fundamental challenges are described below:

- **Problem Formulation:** The problem involves learning a *single and global* statistical model from data stored on millions of remote devices. The main goal is to minimize the global function, which is,

$$min\ F(w),\ where\ F(w) := \sum_{k=1}^{m} p_k F_k(w) \tag{1.1}$$

Here m is the total number of devices, F_k is the local objective function for the kth device, and p_k species the relative impact of each devices on the global model with $p_k > 0$ and $\sum_{k=1}^{m} p_k = 1$.

- **Expensive Communication:** As Federated learning comprises millions of devices, the communication in the network can be prolonged. It is costly as compared to traditional local center training. Therefore, to prevent this expensive communication, it is necessary to create more efficient communication methods that iteratively send training updates in a small message rather than sending the entire progress in one go.

- **System Heterogeneity:** Large variety of system configurations in mobile devices found these days pose a fundamental problem to Federated Learning training. Storage, computational and communication capabilities of devices present in the federated network affects due to variability in hardware(CPU, Memory), network configurations (5G, 4G, 3G), and battery level. These different characteristics cause issues like stragglers and fault-tolerance more significantly than the traditional data centers.

- **Privacy Concerns:** Although federated learning is mainly known for its privacy features. But sometimes, communication between mobile devices and cloud servers while uploading the model reveals sensitive information to third parties or data servers. Also, there is a slight possibility to reverse engineer the model and gather user-sensitive information from it. That's why it is always preferred to communicate in an encrypted way to prevent Man-in-the-middle attacks.

- **Statistical Heterogeneity:** One of the principal fundamental challenges of federated is Statistical Heterogeneity. It can be defined as, "When devices frequently generate and collect data in a *non-identically distributed* manner across the federated network". The number of data points across the devices varies significantly, and it is difficult to capture the underlying relationship and associated distribution. This paradigm violates the I.I.D assumptions and increases the complexity of the algorithm. The next sections detail more information about Statistical Heterogeneity.

1.3 Statistical Heterogeneity

Statistical or Data Heterogeneity means the variability in the dataset. On the other hand, homogeneity means the data is the same. In the field of machine learning, data heterogeneity is addressed as the major concept before starting any project. Ideally, the dataset one is using should be properly and equally distributed among all the classes.

In the case of centralized machine learning, all the modeling and analysis are done on the assumption that data is heterogeneous. Whereas in the case of federated learning, all the data resides on the geographically distributed remote devices. Researchers and companies have no way to find the

heterogeneity of the user's data because of privacy reasons. That's why statistical heterogeneity is the major fundamental problem in the field of federated learning.

Let's understand an example where federated learning suffers from the problem of statistical heterogeneity. Suppose there are 2 users and 3 classes of images - Bottle, Cup, Dog. 1^{st} user clicks 9 images, including 5 of a bottle, 2 of the cup, and 2 of dog. On the other hand, 2^{nd} user clicks 8 images, including 1 of a bottle, 4 of a cup, and 3 of dog. If it is a case of centralized machine learning, then there will be 17 images combined, including 6 images of bottles, 6 images of cups, and 5 images of dogs. All 3 classes will have a similar number of data points, and the model learns very effectively from this dataset. But in the case of federated learning, each user has different data points for each class. Researchers and companies don't know which users have a low number of images from which class. This will badly affect the model training, and their final contribution matters less. So, we say that user datasets are *non-identically distributed*.

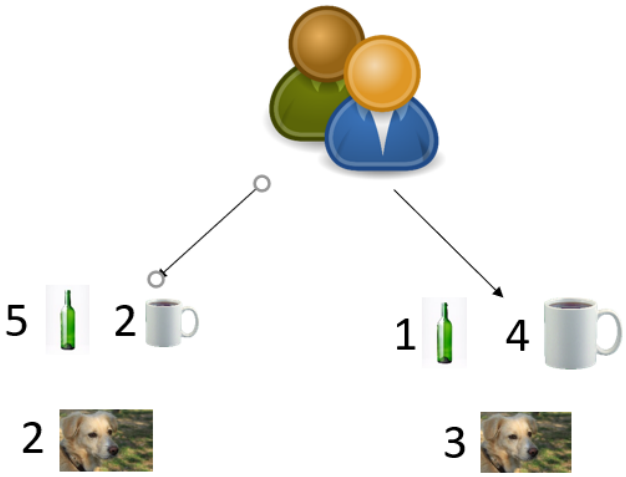


Figure 1.2: Example of problem in statistical heterogeneity

Chapter 2

Our Method

This chapter will detail our proposed method of Image Augmentation and its implementation in a very detailed manner. This chapter will also explain Google's FedAvg, and some other previous work too.

2.1 Previous Work

During this project, we also looked at the previously published work done on 'federated learning given heterogeneous environment' and how it will affect our current path. This section will detail some of them, including FedAvg introduced by Google and Inverse Distance Aggregation.

2.1.1 Google FedAvg

In 2016, when Google introduced the concept of federated learning, they also introduce the FedAvg [5] algorithm, which is responsible for the convergence of the model. Though FedAvg is not very promising in delivering the results, this algorithm is always accounted as baselines for most of the research papers. Please find the Federated Averaging algorithm on the next page.

w_0	inital global model
k	index of K client
B	local minibatch size
η	learning rate
E	local epochs
w_{t+1}^k	local model update
w_{t+1}	global model update

Algorithm 1: FederatedAveraging

Server executes:

initialize w_0

for *each round* $t = 1, 2, ..$ **do**
  $S_t \leftarrow$ random set of m clients
  for *each client* $k \in S_t$ ***in parallel*** **do**
    $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
  end
  $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$
end

ClientUpdate(k, w): //Run on client k

for *each local epoch* i *from 1 to* E **do**
  $w \leftarrow w - \eta \nabla l(w; B)$
end

return w to server

2.1.2 Inverse Distance Aggregation

In Aug 2020, researchers from Technical University of Munich, John Hopkins University, Imperial College London proposed a new averaging algorithm that outperforms the FedAvg. They proposed a method of Inverse Distance Aggregation (IDA) [10], a novel adaptive weighting approach for clients based on meta-information, which handles unbalanced and non-iid data. The core of this method is to calculate the coefficient α_k, which is based on the inverse distance between each client parameter and the average model of all clients. This helps to reject or weigh models who are poisoning.
They used l_1 norm as a metric to calculate the distance between clients weight w_k and averaged weight w_{avg} as,

$$\alpha_k = \frac{1}{Z} \parallel w_{Avg}^{t-1} - w_{k\,t-1} \parallel^{-1}, \tag{2.1}$$

where Z = $\sum_{k \in K} \parallel w_{Avg}^{t-1} - w_{k\,t-1} \parallel^{-1}$ is a normalization factor.

2.2 Our Method - Image Augmentation

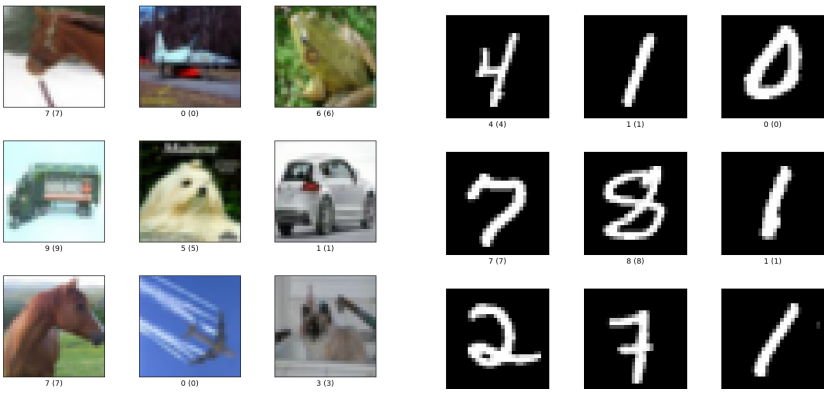
Most of the methods and techniques out there are based on modifying or altering the averaging algorithm. But, we proposed a new method with a different approach that directly tackles the Non-IIDness nature of the dataset. We proposed to use Image augmenting techniques. CIFAR10[11] and MNIST[12] image dataset is used for this research. The core part of this method is to increase the

samples for the class which has fewer images(minority class) using image augmenting techniques without sharing the information amongst the users. We will add 5-6 unique augmented versions of each minority class sample image to the user's dataset. This will help to cure the Non-IID data distribution, as every class will have a similar number of data points across their local data. This will result in efficient model learning, and every user can contribute effectively towards the global model.

2.2.1 Datasets Used

CIFAR10: This image dataset contain around 60,000 32X32 images in 10 classes,i.e, 6000 images per class. 50,000 images in the training set and 10,000 images in the testing set. Classes include airplanes, automobiles, bird, cat, deer, dog, frog, horse, ship, truck. All the classes are completely mutually exclusive. In this research, for training, we didn't use all the images from the training set. Fig2.1(a) shows the sample images from the CIFAR10 dataset.

MNIST: This dataset contains the handwritten digits in 28X28 pixel image format. It contains around 70,000 images in 10 classes (0-9). This dataset is the subset of the EMNIST image dataset, which contains around 6 different big classes, including letters and digits. The training set contains around 60,000 images, and the testing set contains around 10,000 images. Fig2.1(b) shows the sample images from the MNIST dataset.



(a) CIFAR10 (b) MNIST

Figure 2.1: Images from dataset

2.2.2 Image Augmentation Techniques

Image augmentation is a prevalent concept among the machine learning community, not only in image classification but also in data science. Apart from the fact that both image and data augmentation helps to handle the imbalanced dataset, it also prevents overfitting and generalizes the model better. For data augmentation, some of the algorithms are Synthetic Minority Oversampling Technique

(SMOTE) and Near Miss Algorithm. For image augmentation, some of them include Sheering, Rotation, Perspective, etc. Techniques like Neural Style Transfer, GAN, Adversarial training are some of the advanced image augmentation methods. Pytorch framework *Torchvision* and OpenCV are used for augmenting purposes. Image augmentation techniques used in this research include:

- **Random Perspective:** This will change the perspective of the given image with a given probability. We have set its parameters to its default values.
  API to call this function: *torchvision.transforms.RandomPerspective()*
- **Random Horizontal Flip:** This transformation will horizontally flip the given image. It will accept the probability parameter, which is the probability of the image being flipped, and this has been set to 0.5. This transformation was not used in the case of MNIST, as it will confuse our model with the flipped digits.
  API to call this function: *torchvision.transforms.RandomHorizontalFlip(p=0.5)*
- **Random Rotation:** It will rotate the given image by the given angle. It accepts the angle as its parameter, in our case, we have set this range to (-45,45) degrees and set all other parameters to its default values. This transformation is also not used in the case of MNIST, as it will confuse our model with the rotated digits.
  API to call this function: *torchvision.transforms.RandomRotation(degrees=(-45,45))*
- **Blur:** This transformation is achieved by passing the given image through the low pass filter kernel. It is beneficial in removing the high-frequency content like noises and edges. It accepts the kernel size as its parameter, in our case, we have set this to 2X2.
  API to call this function: *cv2.blur(img,(ksize))*

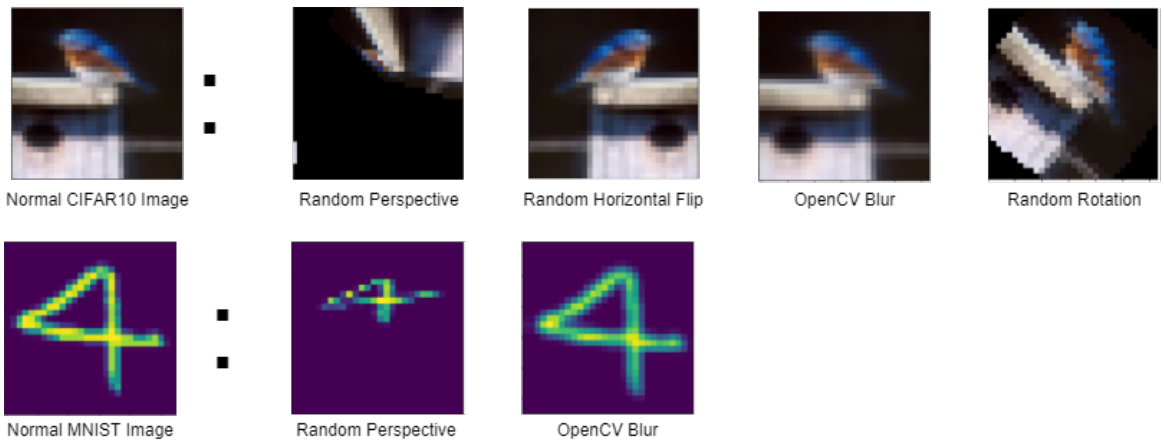


Figure 2.2: Augmented versions of sample images

2.2.3 Method Flowchart

Initially, we will be distributing the whole dataset into the smaller dataset according to a number of the user. For example, if we have 10 users, the whole CIFAR10 dataset will be distributed into the 10 smaller unique datasets. Each user will have only 2 randomly selected majority classes (a class that has a maximum number of images) and 8 minority classes. The majority class samples will form the p%, and minority classes will form the remaining 1-p% of the dataset. This p factor is the deciding component that how many samples each user will have from each class. This *p* value ranges from 0.1-0.99. After distributing the user's dataset, we will process them for better and efficient training, like transforming them into batches and encode their labels into the binary form. We will also create 2 test batches from the testing set to validate the model. We will start training the global and local models. Global models are the models that companies or researchers deploy. Local models are models that user trains on their devices. After obtaining the results, we will be applying the image augmenting techniques to the user's dataset. Again we will train global and local models and obtain their results. Fig2.3 depicts the flowchart of our method.

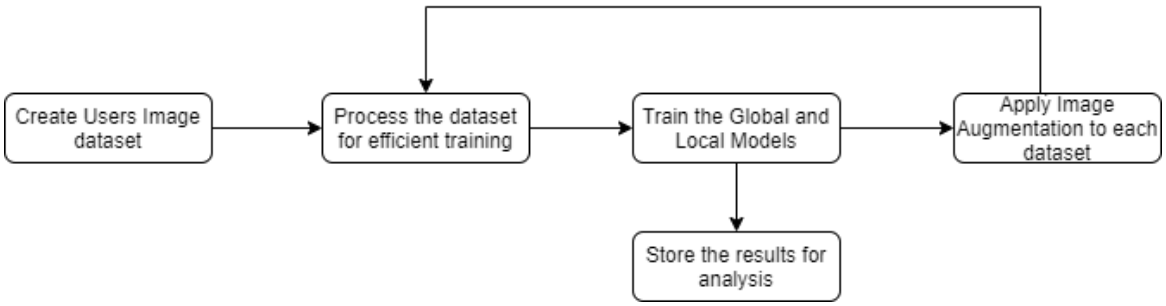


Figure 2.3: Method Flowchart

2.3 Implementation

In this section, we will explain our main code functions in a very explicit manner. The implementation of this code is based on the tutorial [13] present on the medium blogging website.

```
import numpy as np

def sampling(dataset, num_users, p):

    idxs = np.arange(len(dataset),dtype=int)
    labels = np.array(dataset.targets)
    label_list = np.unique(dataset.targets)

    # sort labels
```

```
    idxs_labels = np.vstack((idxs, labels))
    idxs_labels = idxs_labels[:,idxs_labels[1,:].argsort()]
    #print(idxs_labels)
    idxs = idxs_labels[0,:]
    idxs = idxs.astype(int)
    n_data=int(len(dataset)/(2*num_users))
    dict_users = {i: np.array([], dtype='int64') for i in range(num_users)}

    #Sample majority class for each user
    user_majority_labels = []
    for i in range(num_users):
        majority_labels = np.random.choice(label_list, 2, replace = False)
        #2 represent the numbers of majority classes each user will have
        user_majority_labels.append(majority_labels)
        print(i,'\t',majority_labels)
        majority_label_idxs = (majority_labels[0] == labels[idxs])| (majority_labels[1] == labels[idxs])

        sub_data_idxs = np.random.choice(idxs[majority_label_idxs], int(p*n_data), replace = False)

        dict_users[i] = np.concatenate((dict_users[i],sub_data_idxs))
        idxs = np.array(list(set(idxs) - set(sub_data_idxs)))

        #assigning minor classes to each client
    if(p < 1.0):
        for i in range(num_users):
            majority_labels = user_majority_labels[i]

            non_majority_label_idxs = (majority_labels[0] != labels[idxs]) |
                (majority_labels[1] != labels[idxs])

            sub_data_idxs = np.random.choice(idxs[non_majority_label_idxs], int((1-p)*n_data),
                replace = False)

            dict_users[i] = np.concatenate((dict_users[i], sub_data_idxs))
            idxs = np.array(list(set(idxs) - set(sub_data_idxs)))
    idx=int((p)*n_data)

    return dict_users,idx
```

This code is used to sample the whole dataset into a smaller set for users. This code will add only 2 randomly selected majority classes, and 8 other remaining classes will be minority classes. It accepts *dataset, num_users, p* as their parameters and returns the dictionary type *dict_users*, which stores the image indexes for each user. This function will select random image indexes from the user's 2 majority and 8 minority classes and keep deleting those selected indexes from the whole dataset to maintain the uniqueness among the users. If the *p* value is less than 1, then only it will add minority class image samples. This way, we maintain the non-iid distribution in the users' datasets. This function will also return *idx*, the index where the majority class images end, it will help us later to apply image augmentation.

Note: For the research purpose, we have used this *idx* approach to detect where minority class samples start in the dataset. In real life, this *idx* value can't be obtained, in that case, counting the samples belongings to each class from the user dataset and finding the ratio of max sample class to other class samples will be the best approach. With this method, detecting our minority classes and apply image augmentation to them will be easy.

In the next step, we obtained the images from the whole dataset using indexes and converted them into batches, and encoded their label into binary form. A similar process is used to create 2 test batches for testing and validation purpose. Due to resource limitations, the original image size has been resized to 12X12X3 pixels. These things will help to train the model fast and efficiently.

```
def weight_scalling_factor(clients_trn_data, client_name):
    client_names = list(clients_trn_data.keys())
    #get the bs
    bs = list(clients_trn_data[client_name])[0][0].shape[0]
    #first calculate the total training data points across clients
    global_count = sum([tf.data.experimental.cardinality(clients_trn_data[client_name])
        .numpy() for client_name in client_names])*bs
    # get the total number of data points held by a client
    local_count = tf.data.experimental.cardinality(clients_trn_data[client_name]).numpy()*bs
    return local_count/global_count


def scale_model_weights(weight, scalar):
    '''function for scaling a models weights'''
    weight_final = []
    steps = len(weight)
    for i in range(steps):
        weight_final.append(scalar * weight[i])
    return weight_final



def sum_scaled_weights(scaled_weight_list):
    '''Return the sum of the listed scaled weights. The is equivalent to scaled avg of the weights'''
    avg_grad = list()
    #get the average grad accross all client gradients
    for grad_list_tuple in zip(*scaled_weight_list):
        layer_mean = tf.math.reduce_sum(grad_list_tuple, axis=0)
        avg_grad.append(layer_mean)

    return avg_grad


def test_model(X_test, Y_test,  model, comm_round):
    cce = tf.keras.losses.CategoricalCrossentropy(from_logits=True)
    #logits = model.predict(X_test, batch_size=100)
    logits = model.predict(X_test)
    loss = cce(Y_test, logits)
```

```
    acc = accuracy_score(tf.argmax(logits, axis=1), tf.argmax(Y_test, axis=1))
    print('comm_round: {} | global_acc: {:.3%} | global_loss: {}'.format(comm_round, acc, loss))
    return acc, loss
```

These are some important utility functions that help during model training. The first function ***weight_scalling_factor*** helps to count the number of clients presented during training. It will first count all the training points across clients and divides them by the number of client's data points. The second function ***scale_model_weights*** is used to multiply the model weights obtained from the client after training by the weight scaling factor calculated earlier. Both of these functions are used to scale the model weights by the factor of the number of clients.

The third function ***sum_scaled_weights*** is used to sum all the scaled model weights and is equivalent to the scaled averaged of the weights. The fourth function ***test_model*** is used to test and validate our model against the test set. It returns the accuracy and loss of the model, it helps us to prevent overfitting.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_16 (Conv2D)           (None, 12, 12, 32)        896
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 12, 12, 32)        9248
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 12, 12, 64)        18496
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 12, 12, 64)        36928
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 6, 6, 128)         73856
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 6, 6, 128)         147584
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 3, 3, 128)         0
_________________________________________________________________
dropout_4 (Dropout)          (None, 3, 3, 128)         0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 3, 3, 256)         295168
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 3, 3, 256)         590080
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 1, 1, 256)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 500)               128500
_________________________________________________________________
dropout_5 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               64128
_________________________________________________________________
dropout_6 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
```

```
=================================================================
Total params: 1,366,174
Trainable params: 1,366,174
_________________________________________________________________
```

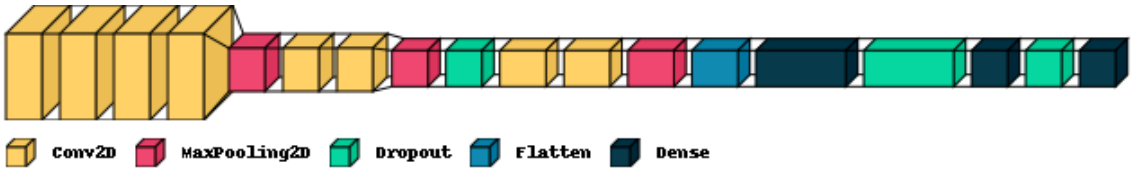


Figure 2.4: Layout of our model

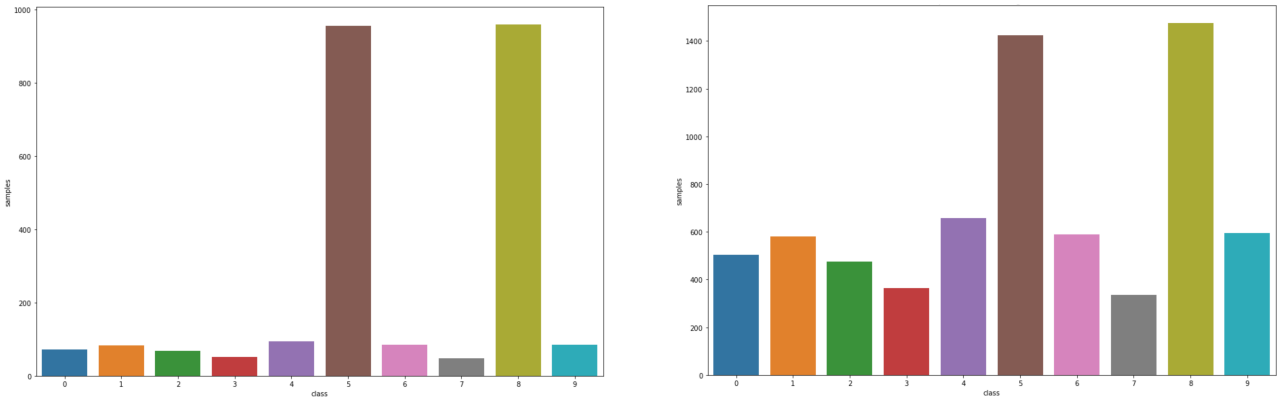
The previous figure depicts our model *sequential_1* summary that we used during training. We used a sequential Convolution Neural Network, which comprises 8 convolution layers. We used *flatten* layer to convert the 3D matrix into the 1D array. In the end, we used a *dense* layer to convert the large array of 64K into the smaller array of 10, which defines our classes. The model contains the 1.3M parameters, and all of them are trainable. We used the *softmax* activation function and *adam* function for optimization purposes.

```
#initialize global model
smlp_global = models.SimpleMLP()
global_model = smlp_global.build(10,image_shape)

comm_round=10
#commence global training loop
for comm_round in range(comm_round):
    # get the global model's weights - will serve as the initial weights for all local models
    global_weights = global_model.get_weights()

    #initial list to collect local model weights after scaling
    scaled_local_weight_list = list()
    #randomize client data - using keys
    client_names= list(clients_batched.keys())
    random.shuffle(client_names)

    #loop through each client and create new local model
    for client in client_names:
        smlp_local = models.SimpleMLP()
        local_model = smlp_local.build(10,image_shape)
        #set local model weight to the weight of the global model
        local_model.set_weights(global_weights)
        #fit local model with client's data
        local_model.fit(clients_batched[client], epochs=10, verbose=0,validation_data=test_batched_1)

        #scale the model weights and add to list
        scaling_factor = weight_scalling_factor(clients_batched, client)
```

```
        scaled_weights = scale_model_weights(local_model.get_weights(), scaling_factor)
        scaled_local_weight_list.append(scaled_weights)

        #clear session to free memory after each communication round
        K.clear_session()

    #to get the average over all the local model, we simply take the sum of the scaled weights
    average_weights = sum_scaled_weights(scaled_local_weight_list)

    #update global model
    global_model.set_weights(average_weights)
    model_json = global_model.to_json()
    with open("model.json", "w") as json_file:
        json_file.write(model_json)
    # serialize weights to HDF5
    global_model.save_weights("model.h5")
    print("Saved model to disk")

    #test global model and print out metrics after each communications round
    for(X_test, Y_test) in test_batched_2:
        global_acc, global_loss = test_model(X_test, Y_test, global_model, comm_round)
```

This code is used to train the model across the federated network. Initially, *global_model* is created using the *SimpleMLP* class instance. Here, *comm_round* is the communication round which is the number of times the global model will communicate with the local models. Its value is governed by the type of dataset used, like 15 will be enough in the case of MNIST and around 50-60 in CIFAR10. Then we initialize the local models for each client and set their model weights equal to global model weights. After that, each client will train their local model and validate it against the test batch. After completion of local model training, we scale their weights using *weight_scalling_factor*. Then, all the local weights will be averaged, and update the global model weights. The global model will be tested using a test batch, and their performance will be stored for later analysis. After then, we deploy the updated global model to all the clients. Again, clients will train the local models and send their updated weights. This code also saves the updated global model in case of failure and helps to resume the training from the last state. This cycle will keep running until we achieved some good results or communication round ends.

After completing the global and local model training, we apply image augmentation to the client's dataset. And then, again, global and local models will be trained using the similar approach described above, and their performance will be stored. Fig2.5 shows the number of image samples per class before and after augmentation. In Fig2.5(a), the user has 2500 images, and after applying image augmentation to his dataset, the images increased to 7000, as shown in Fig2.5(b). Due to the resource limitation, we have added only a fewer number of augmented images.



(a) Before Augmentation

(b) After Augmentation

Figure 2.5: Number of Image Samples/Class

Chapter 3

Results and Conclusion

This chapter will include the results and analysis of our method. It also details the future work or some methods to improve the accuracy of this method.

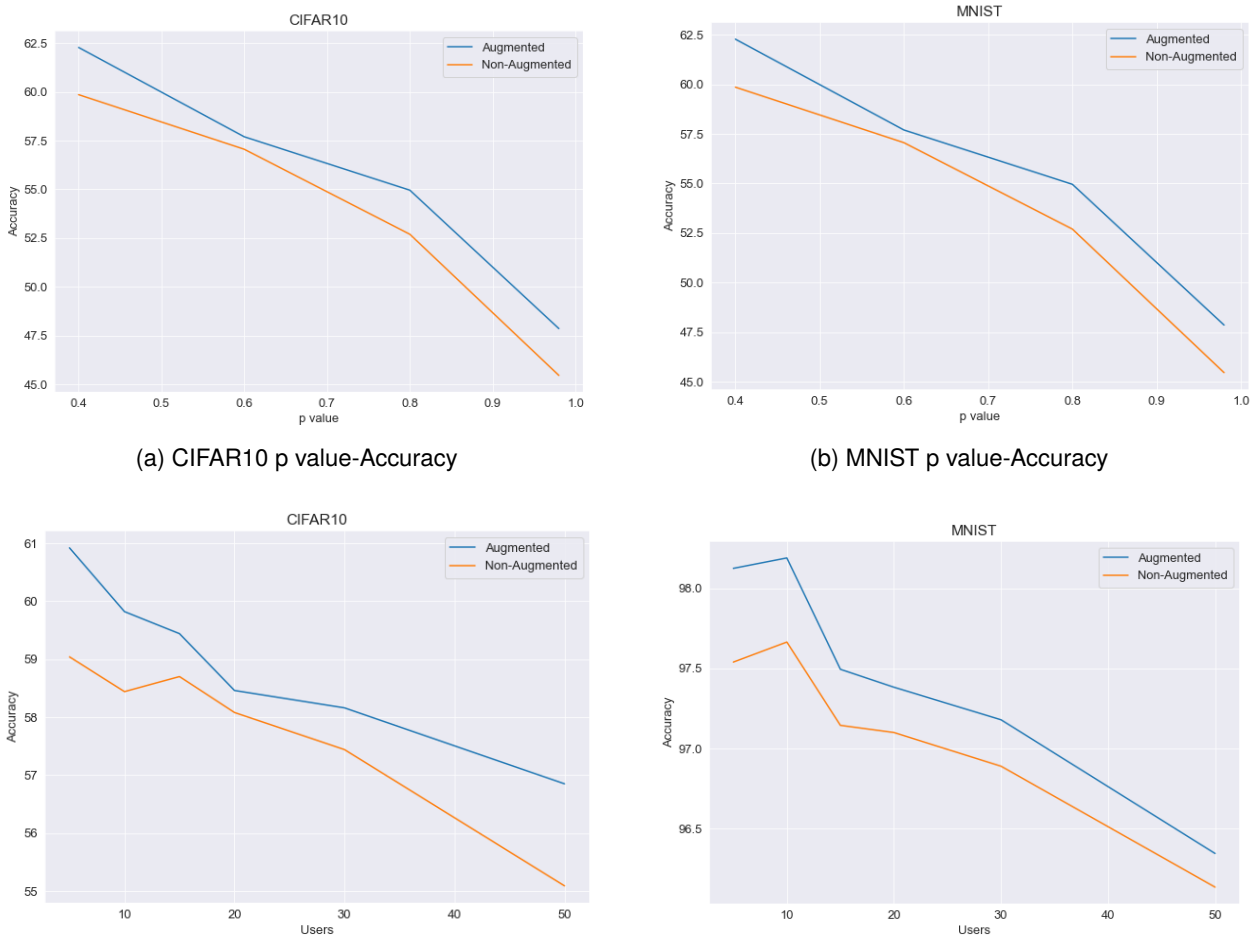
3.1 Results

During model training, we stored the performance of both the training without and with Image Augmentation. There are parameters on which we analyzed our method are as follows;

- ***p value***, it defines how much portion of the user's dataset will contain majority class samples. Its value ranges from 1 - 0.99, and it cannot be equal to 1; otherwise, the user will contain only the majority of class samples. We checked this parameter on values - 0.4, 0.6, 0.8, 0.99.
- ***# of users***, We checked our method on users ranging from 10-50. Originally, the federated network will consist of millions of users. But to due to resource limitation, we have constrained it to only 50.
- ***comm_round***, in-case of CIFAR10, we used more than 45-60 and 10-15 in-case of MNIST. This parameter is crucial with regard to accuracy. The larger the *comm_round* value, the higher the accuracy will be.

Fig3.1 shows the result of our method. This figure consists of 4 sub-figure which plot the accuracy vs. different parameters. Blue-line exhibits the Image Augmentation method, and Orange-line denotes accuracy for training without image augmentation.
In Fig3.1(a,b), a plot was drawn between p-value and Accuracy for both CIFAR10 and MNIST datasets. For any p values when the number of users=10, our method records higher accuracy than Google's FedAvg.
In Fig3.1(c,d), a plot was drawn between the number of users and Accuracy for both CIFAR10 and

MNIST datasets. The plots reveal that for any given number of users and using 0.7 as p value, and our method outperforms the Google FedAvg.



(a) CIFAR10 p value-Accuracy

(b) MNIST p value-Accuracy

(c) CIFAR10 Users-Accuracy

(d) MNIST Users-Accuracy

Figure 3.1: Results

3.2 Analysis

Some analysis can be concluded from this research and its results which is as follows;

1. Downward slope can be seen, which means that an increasing number of users will lower accuracy. And we can say that,

$$NumberofUsers \propto \frac{1}{Accuracy}$$

2. Falling slope can be observed, and this also means that increasing *p* value will also result in lower accuracy. And we can say that,

$$pvalue \propto \frac{1}{Accuracy}$$

3. *p* value cannot is equal to 1, otherwise, the user will not possess minority class samples. This will cause a problem during training.

4. Our method will increase the accuracy by 4% for any p value and 1.7% for any number of users.

3.3 Future Work

Our research has a lot of room for improvement. There are some of the things that we can implement in our project to increase accuracy and decrease loss. Limitation to the resources provided for this research leads to future work. These things are as follows:

- **More number of users**, Originally federated network consists of millions of user, but in our research, we have constraint our users to only 50. We need to check that how our method works when there are many users.

- **Different *p* value**, we need to see how our method reacts to different *p* values and especially values close to 1.

- **Transfer Learning**, it can be defined as a "method where a model developed for a task is reused as the starting point for a model on a second task." It is an up-and-coming technique and helps companies and researchers to save time and resources. We can use an advanced pre-trained model instead of using the basic CNN model. Some of the examples are VGG16, GoogLeNet, Xception, etc.

- **Different Dataset**, we have only applied our method to MNIST and CIFAR10 image dataset. But we need to check our method on lots of heterogeneous datasets present on the internet. It helps to understand more about image augmentation.

- **Advanced Image augmentation**, as it has been already discussed in the Image Augmentation Techniques section. These methods include Adversarial Training, Generative Adversarial Networks, Style Transfer, and using Reinforcement learning to search through the space of augmentation possibilities. Applying these methods will helps to improve model learning.

- **Code Optimization**, our code uses 2 different framework - Tensorflow and Pytorch. Using different frameworks uses too much memory and takes time to process. So, there is room for code optimization, which helps in training heavy models without utilizing too much RAM and time.

Bibliography

[1] Open Data Science. What is federated learning?, 2020.

[2] Alexandra Lheureux, Katarina Grolinger, Hany F Elyamany, and Miriam AM Capretz. Machine learning with big data: Challenges and approaches. *Ieee Access*, 5:7776–7797, 2017.

[3] Catalin Cimpanu. Adobe left 7.5 million creative cloud user records exposed online, 2019.

[4] Emma Bowman. After data breach exposes 530 million, facebook says it will not notify users, 2021.

[5] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[6] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.

[7] Kyle Wiggers. Major pharma companies, including novartis and merck, build federated learning platform for drug discovery, 2020.

[8] Alex Galakatos, Andrew Crotty, and Tim Kraska. Distributed machine learning, 2018.

[9] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[10] Yousef Yeganeh, Azade Farshad, Nassir Navab, and Shadi Albarqouni. Inverse distance aggregation for federated learning with non-iid data. In *Domain Adaptation and Representation Transfer, and Distributed and Collaborative Learning*, pages 150–159. Springer, 2020.

[11] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[12] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017.

[13] Saheed Tijani. Federated learning: A step by step implementation in tensorflow, 2020.