© Copyright 2022

Zhiyang Zhou

Facial Recognition on Android Devices with Convolutional Neural Networks and Federated Learning

Zhiyang Zhou

A capstone project

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2022

Committee:

Afra Mashhadi, Chair

Johnny Lin

Kelvin Sung

Geethapriya Thamilarasu

Program Authorized to Offer Degree:

Computing and Software Systems

University of Washington

Abstract

Facial Recognition on Android Devices with Convolutional Neural Networks and Federated Learning

Zhiyang Zhou

Chair of the Supervisory Committee:
Assistant Professor Afra Mashhadi
Computing and Software Systems

Machine Learning (ML) and Artificial Intelligence (AI) are widely applied in many modern services and products we use. Facial Recognition (FR) is a powerful ML application that has been used extensively in various fields. Traditionally, the models are trained on photos gathered from the World Wide Web (WWW), and they are often biased towards celebrities and the Caucasian population. Centralized Learning (CL), one of the most popular training techniques, requires all data to be on the central server to train ML models. However, it comes with additional privacy concerns as the server takes ownership of end-user data. In this project, we first use Convolutional Neural Networks (CNN) to develop an FR model that can classify 7 demographic groups using the FairFace image dataset. This has a more balanced and diverse distribution of ordinary face images across the racial groups. To further extend the training accessibility and protect sensitive personal data, we propose a novel Federated Learning (FL) system using Flower as the backend and Android phones as edge devices. These pre-trained models are initially converted to TensorFlow Lite models, which are then deployed to each Android phone to continue learning on-device from additional subsets of FairFace. Training takes place in real-time and only the weights are communicated to the server for model aggregation, thus separating user data from the server. In our experiments, we explore various centralized model architectures to achieve an initial accuracy of 52.9%, which is lightweight enough to continue improving to 68.6% in the Federated Learning environment. Application requirements on Android are also measured to validate the feasibility of our approach in terms of CPU, memory, and energy usage. As for future work, we hope the system can be scaled to enable training across thousands of devices and have a filtering algorithm to counter adversarial attacks.

TABLE OF CONTENTS

List of Figures iv

List of Tables v

Chapter 1. Introduction 7

1.1 Overview and Motivation 7

1.2 Proposed Work 8

1.3 Report Structure 11

Chapter 2. Related Works 12

2.1 Centralized Training with CNN 12

2.1.1 How CNN Models Learn from Images 12

2.1.2 Factors That Affect the Model Learning 13

2.2 CNN Applications with Facial Data 14

2.3 TensorFlow Lite in Decentralized and Federated Learning 15

2.4 Background Summary and Project Motivation 17

Chapter 3. Methodology 18

3.1 Project Overview 18

3.2 Dataset 19

3.2.1 Understanding the Content 19

3.2.2 Preprocessing 21

3.2.3 Practical Challenges with Racial Classification 23

3.3 Stage 1: Preparing CNN Models with Centralized Training 23
3.3.1 VGGFace Version 1.....24
3.3.2 VGGFace Version 2.....27
3.3.3 ResNet50 28
3.3.4 MobileNetV2.....29
3.4 Stage 2: Converting Trained Models to TensorFlow Lite 30
3.4.1 TF Lite Optimization 30
3.4.2 Conversion Process.....31
3.5 Stage 3: Deploying TF Lite Models to Federated Learning 33
3.5.1 Flower Backend.....33
3.5.2 Server 34
3.5.3 Client.....35
3.5.4 Communication 36
3.6 Evaluation Metrics.....37
Chapter 4. Experiment Results and Evaluation 38
4.1 Stage 1: CNN Model Training 38
4.1.1 VGGFace Version 1.....38
4.1.2 VGGFace Version 2.....40
4.1.3 ResNet50 40
4.1.4 MobileNetV2.....41
4.1.5 Model Comparison and Analysis 42
4.2 Stage 2: TF Lite Conversion 43
4.3 Stage 3: Flower Federated Training 44

4.3.1 VGGFace 45
4.3.2 ResNet50 47
4.3.3 MobileNetV2 48
4.3.4 Test Result Comparison 50
4.4 Stage-By-Stage Evaluation 51
Chapter 5. Conclusion 53
Chapter 6. Future Work 54
Bibliography 55

LIST OF FIGURES

Figure 1. Project workflow overview..... 18
Figure 2. Original folder structure of the dataset that contains only two folders: train and validation..... 20
Figure 3. A side-to-side comparison between the two different Padding values. 21
Figure 4. Preprocessed folder structure for the image dataset. Each race group contains only images with the same label. 22
Figure 5. Comparison between the official *TF Lite* library (Left) and Flower *tfltransfer* library (Right)..... 31
Figure 6. Illustration of layer manipulation to combine Base Model with Head Model for Resnet50..... 32
Figure 7. System Architecture of Flower Android framework..... 33
Figure 8. Flower Android Application GUI. 36
Figure 9. VGGFace Version 1 training and validation convergence plots. 39
Figure 10. VGGFace partially freezing training and validation convergence plots. 40
Figure 11. ResNet50 training and validation convergence plots. 41
Figure 12. MobileNetV2 training and validation convergence plots. 42
Figure 13. Flower Android app GUI during training. 44
Figure 14. VGGFace Test Accuracy plot. 45
Figure 15. VGGFace Loss plot. 46
Figure 16. Android resource usage during VGGFace training. 47
Figure 17. MobileNetV2 Test Accuracy plot for Android devices. 48
Figure 18. MobileNetV2 Loss plot for Android devices. 49
Figure 19. Android resource usage monitoring during MobileNetV2 training. 50

LIST OF TABLES

Table 1. Popular public face datasets that are more biased towards the Caucasian population [32, p. 5]. 19
Table 2. Overview of the original image distribution in the dataset. 20
Table 3. Overview of image distribution per race for the learning environments. 22
Table 4. VGGFace model architecture with new head layers. 24
Table 5. VGGFace and ResNet50 compile and training hyperparameters. 26
Table 6. VGGFace model architecture without new head layers. 27
Table 7. MobileNetV2 model compile settings and training hyperparameters. 29
Table 8. Server Specification. 34
Table 9. Android Device Specification. 35
Table 10. CNN model comparison. 42
Table 11. Overview of model sizes before and after the conversion. 43
Table 12. Model test results in Flower Federated. 51

ACKNOWLEDGEMENTS

I would like to first express my sincere gratitude to Dr. Afra Mashhadi, the chair of my Supervisory Committee, who has provided me with an incredible amount of support throughout all stages of my master study. Her deep knowledge, constructive mentorship, and immense research drive has given me crucial guidance in the completion of this capstone project. I would also like to thank my committee members, Dr. Johnny Lin, Dr. Kelvin Sung, and Dr. Geethapriya Thamilarasu, for their tremendous support, timely responses, and insightful feedback. Each of these individuals plays an important role in my professional and personal growth.

Chapter 1. INTRODUCTION

1.1 Overview and Motivation

The rapid development of Machine Learning (ML) has enabled the advancement of technological tools to create impactful renovations in the world. It helps people solve complex problems in various fields. Content recommendation, pattern prediction, business insights, smart driving, medical diagnosis, and other ML-based applications are making revolutionary changes to society [1]. Each of these examples belongs to one of the three following categories: Reinforcement Learning (RL), Unsupervised Learning (UL), and Supervised Learning (SL) [2]. In particular, SL learns from existing input and output data pairs, identifies patterns, and associates features with specific groups. After an SL model is trained, it can later be used to predict the label of previously unseen samples from the same domain as the training data.

One of the examples of SL is image classification. Typically, an image can contain thousands or millions of pixels, where each individual pixel contains light information received at that image coordinate. For colored images, a pixel describes a combination of the intensity of three color components: Red, Green, and Blue [3]. While images often contain a large volume of data, it is also difficult to have an automated method to extract useful information from the pixels. ML, particularly Convolutional Neural Network (CNN), made it possible by mimicking the way human perceives visual data. Instead of considering one individual pixel at a time, we can train a ML model to learn to locate groups of pixels that potentially carry useful information and remember what the patterns look like. By associating patterns to specific labels in the SL setting, a model can perform image classification tasks, such as identifying if a patient's X-ray scan contains cancer cells or recognizing if a face image is associated with a particular group [4]. Evidently, using ML

to process image data has brought many benefits to our life.

There are also many problems and challenges in the field of ML. Facial Recognition (FR) is a ML application that is typically used to extract information from face images and associate it with a label. However, it falls short in many cases. In recent years such systems have been used by law enforcement to help identify suspects from surveillance recordings. Many FR models have shown evidence of racial profiling and are often biased towards the darker-skinned population [5]. This is generally because the training dataset has a poor presentation of the actual demographic background, producing biased knowledge that is passed to the ML models.

Furthermore, ML also faces a practical challenge when people of the same ethnicity can look drastically different, and people of different ethnicities can look similar. This can be confusing to ML models as they attempt to associate images with the correct labels. Realistically, it is a difficult task to make accurate racial classifications for not only ML models but also human eyes.

Most FR models are trained in a Centralized Learning (CL) setting, where the data and model are required to exist within the same host machine during the training process [6]. During the preparation of a training dataset, demographic labels for each image are manually created based on the user information stored on the server. Such sensitive demographic information is not only shared with the server but also can be publicly accessible if such curated face datasets are released to the internet. Centralized demographic labeling creates an increasing concern for user privacy.

1.2 PROPOSED WORK

To address these issues, this project first aims to reduce racial biases in the process of creating FR models, which can lead to a more equitable representation of demographic backgrounds. Next, we want to produce a ML model that can classify different ethnicities with a high accuracy, which is

an important metric in measuring performance. Lastly, we want to keep user data on their device separated from the central server throughout the training processing to preserve privacy by creating a Federated Learning (FL) system [7].

There are a few existing solutions that are relevant to this project and can help us achieve the goals mentioned above. First, to combat racial biases it is crucial to consider various factors that can affect training, especially from input data. A ML model trained with a well-diversified dataset is more likely to have a lower level of biases. Demographic background information such as skin color, age, gender, and historical disadvantages all contributes to the overall representation of a face dataset [8].

Many ML techniques can improve model performance and accuracy. One of the most common methods is to increase the input complexity by adding additional datasets or augmenting existing data for training. Second, different model architectures can also lead to different learning results. When there is enough input data, a ML model with more trainable parameters is likely to have better final accuracy. Lastly, fine-tuning the hyperparameters of a trained model properly can usually yield a slightly improved accuracy by a few percentage points.

In terms of preserving user privacy, one way is to set up Decentralized Learning (DL) and enable on-device training. While traditional TensorFlow models are too heavy for edge devices, they can be converted to TensorFlow Lite (TF Lite) models and then deployed to the client-side [9]. Sensitive data is used to train TF Lite on the user device and stays separate from the server. This setup allows users to benefit from ML with protected privacy.

To make even better use of DL, Federated Learning (FL) is a new solution that creates a decentralized yet collaborative ML system [7]. This type of system combines the learning results from each user device and produces an aggregated model with improved overall performance

while making sure the user's private data stays preserved throughout the process. In recent years, TensorFlow Federated [9] and Flower [10] are popular libraries used as the backend of a FL system.

In this project, we carefully examined state-of-the-art face datasets by comparing the number of face images per ethnicity and evaluating the data coverage. After selecting the dataset with the best racial diversity, we used it to train Convolutional Neural Network (CNN) [11] models and tested them with different settings.

We also explored various techniques and observed their effects on the model performance. First, we wanted to increase the input complexity. This can be done by applying pre-trained weights from other facial data sources or augmenting the current dataset. Next, we experimented with different model architectures and layer setups. In the training stage, Transfer Learning (TL) and hyperparameter tuning were utilized to try to further improve the results.

Next, we converted the trained CNN models into TF Lite models and noted the changes mainly in terms of the model size. For on-device training with Android phones, it is especially important to ensure the models are compact, as mobile phones have less processing power, limited storage space, and small memory capacity.

Finally, we created a FL system to test the viability and the performance of the TF Lite models. We measured the accuracy and loss changes after each round of training to verify the models can continue learning. In the meantime, we also monitored the device resource usage such as CPU, memory, and energy, to check if a model is suitable to be deployed on Android.

As a result, we trained with four model architectures under different settings and converted them into TF Lite models. We also created a functional system that supports federated training between the server and clients. These TF Lite models were deployed to the system to test how well

they performed in the Android federated environment. One particular model shows a moderate use of resources and displays an overall accuracy improvement on Android.

1.3 REPORT STRUCTURE

The report structure is as follows: Chapter 2 talks about the previous research that is relevant to the work done in this project. Chapter 3 describes the methodology, system architecture, and the main components in detail. Chapter 4 shows the testing results of the three stages (Centralized Training, TF Lite Conversion, and Federated Training). Chapter 5 is the discussion of how these results are evaluated. Chapter 6 sums up the conclusion we can obtain based on the evaluation and mentions future works that can further improve the system.

Chapter 2. RELATED WORKS

In this chapter, we show more details on how CNN models can be trained with visual data and create applications to analyze facial images. We also discussed the potential to deploy CNN models on edge devices to enable client-side training to form a FL environment.

2.1 CENTRALIZED TRAINING WITH CNN

It requires a large amount of data to properly train a ML model and the learning process can be computationally intensive. With a powerful server that has large storage and memory space, CL is a common methodology to train models efficiently. In this environment, studies have shown that CNN can effectively gain knowledge from images and make classifications for future input. The performance can vary depending on how the model architectures are constructed and how the learning process is set.

2.1.1 *How CNN Models Learn from Images*

CNN is a type of deep Neural Networks that can learn from visual data [11]. A color image consists of three color spaces: Red, Green, and Blue (RGB). When an input image is passed through the convolution layer, a filter is used to scan through the pixels and group them using matrix multiplication. During this process, the original pixels are extracted as high-level features. These features typically go through a pooling layer where the spatial size is reduced. These two layers are placed together repeatedly, and the original input is reduced to a much smaller feature map. Next, the feature map is fed to fully connected layers that forward propagate values satisfying the activation function. Finally, it produces an output layer whose dimension is equal to the number of classes. The output values are compared to the labels by a loss function, which in turn adjusts

the weights through back propagation before the next round of training [12].

By repeating these steps, we can construct and train deep CNN models for image classification tasks. In 2019, the research project led by Abu et al. [13] studied how well CNN models perform classification on 5 types of flower images: roses, daisy, dandelion, sunflower, and tulips. The experiment result showed that the lowest accuracy of the model is 90.585% for Rose prediction while maintaining at least 99.626% accuracy for other flower types.

2.1.2 *Factors That Affect the Model Learning*

There are mainly three factors that can significantly affect the results of CNN model training: input data, model architecture, and hyperparameter.

Typically, the more well-diversified input data a CNN model receives, the more information the model can learn from. The simplest way to add more data points is to use additional datasets that contain images of the same category. On the other hand, if there are limited image sources, data augmentation is a useful technique that can transform existing input with flip, rotation, scale, crop, and translation [14]. Furthermore, if a model is previously trained on a different data source, Transfer Learning (TL) allows the model to reuse the pre-trained weight and enables the continuous learning of new low-level features. Hussain, M et al. [15] explored the viability of applying TL to Inception-v3 [16], which was pre-trained on a base dataset ImageNet. By modifying certain layers of Inception-v3 and retraining it with a different dataset, they showed the model can learn new features with improved accuracy while using less computational memory and time.

Using different model architectures can also determine the training outcome. Sultana et al. [17] compared the performance of 7 different CNN models such as LeNet-5, SENet, and AlexNet.

By manipulating the number of layers and the number of trainable parameters, the models have a final accuracy of various levels even though they are trained on the same dataset (ImageNet [18]). It is important to note that a larger number of trainable parameters generally produces a higher model accuracy. However, this can also lead to model overfitting if there are not enough training images [19].

Hyperparameter tuning is a process of finding the optimal training setting for a ML model to further improve performance. Deep CNN models have hyperparameters such as learning rate, batch size, momentum, and patience for early stopping [20]. One set of optimal hyperparameter values for a CNN model might not work for another model with a different task or architecture. For each problem, we can try to find the best configuration through trial-and-error testing and result examination.

2.2 CNN Applications with Facial Data

Face images are a popular data source that CNN models can process. There are three common applications in this field: Face Detection (FD), Face Verification (FV), and Facial Recognition (FR).

FD is the process of looking for the existence of faces in an image or a video stream. Dlib [21], MTCNN [22], and OpenCV [23] are popular libraries that provide built-in functions to detect faces with CNN-based face detectors. The main idea is to scan through the pixels and detect if any data patterns match with the facial embedding, which is built with previously learned face knowledge. For each candidate, it performs binary classification on whether it is a face or not with a confidence score.

FV is essentially also a binary classification problem. One main difference is that the models need to be pretrained with a particular dataset to gain specific knowledge about the identity of a

person. The dataset should include face images of the same person with different light settings, angles, hairstyles, or facial accessories (if applicable). The wider the coverage of these images, the more information the model can gain. Chen J, Patel V. M., and Chellappa R. [24] explored FV using deep CNN models with pretrained face weights. These models gained new knowledge from training on specific celebrity faces and can then verify if a new face image belongs to one of the celebrities with a mean accuracy of at least 95.92%.

FR is the process of extracting demographic information from face images and it is the most complex of the three. Depending on the goal of the prediction, the models can be trained to learn information over various attributes associated with face images, such as age, gender, pose, race, or expression. In 2018 Cao Q. et al. [25] trained the ResNet-50 [26] architecture with VGGFace2 [25] to measure model performance over pose and age. With over 3.31 million face images, this dataset yields a much better result compared to previous state-of-the-art datasets. In 2019 Kärkkäinen K. and Joo J. [27] further explored the model fairness among the most popular face datasets. They trained ResNet-34 [26] over four race annotations: White, Black, Asian, and Indian. The model shows a 75.4% accuracy in predicting the correct ethnicity, which is more than twenty percentage points lower than a FV task [24]. FR is complicated as it requires a large amount of information to form a good generalization of the racial representation. Not training FR models properly can create demographic bias and a false recognition can lead to severe consequences. Many FR models have been trained with imbalanced datasets and are often biased against people of darker skin colors [28].

2.3 TensorFlow Lite in Decentralized and Federated Learning

Decentralized Learning (DL) enables training on the client-side and gives users more privacy. TensorFlow Lite (TF Lite) [9] is a state-of-the-art library that supports on-device training for DL.

It can reduce the size of deep CNN models and convert them into compact TF Lite models. There are three common optimization techniques for this process: quantization, pruning, and clustering.

Quantization [9] reduces the model weights, often pre-trained, to types with precision points of lower complexity. This technique decreases the size of the model while only causing minor model accuracy changes. Pruning [9] eliminates certain parameters from a trained model while having little impact on the predictive accuracy. It does not change the size of the model, but it allows model compression to be more effective. Clustering [9] also enables better size compression. The process is to put the layer weights into separate clusters and share each cluster centroid value, reducing the number of unique weight values.

These techniques allow CNN models to become lightweight enough for low-performance edge devices such as mobile, microcontrollers, and Linux-based systems. However, the new knowledge gained on one device cannot be shared with others to improve their model performance in a DL environment. Federated Learning (FL) is a type of DL framework that has gained popularity in recent years, which overcomes the knowledge-sharing restriction by allowing two-way communication between the server and the client. This framework increases the overall learning effectiveness while keeping the user data private on their device during the entire training process.

One popular FL framework is TensorFlow Federated (TFF) [9]. TFF offers FL APIs for tasks such as Image Classification and Text Generation. But one major restriction is it currently only supports TF model architectures. In contrast, Flower [10] is another popular FL framework that supports many popular libraries with CNN implementations, such as TF, PyTorch [29], MXNet [30], scikit-learn [31], and TF Lite.

2.4 Background Summary and Project Motivation

CNN is a powerful ML technique that can process image data, identify patterns, extract visual features, and gain knowledge. A trained model can make classification on new images within the same knowledge pool. Facial Recognition, as one of many CNN applications, is a technology with many useful purposes to solve problems in real life. While it has the potential to do even more, FR often contains bias that creates negative impacts. Furthermore, current ML frameworks generally lack privacy assurance when working with sensitive data.

In this project, we used some of these existing ML methodologies for model preparation and developed a novel FL system to address privacy concerns. First, we carefully selected a face dataset with the least racial biases. Next, we constructed four different CNN model architectures and applied various tuning techniques. These models were then converted to TF Lite models and deployed to our FL system implemented with Flower, which utilizes client-side training to keep user data separate from the server. Finally, model performance and Android resource usage were measured to evaluate the level of success we achieved.

Chapter 3. METHODOLOGY

3.1 PROJECT OVERVIEW

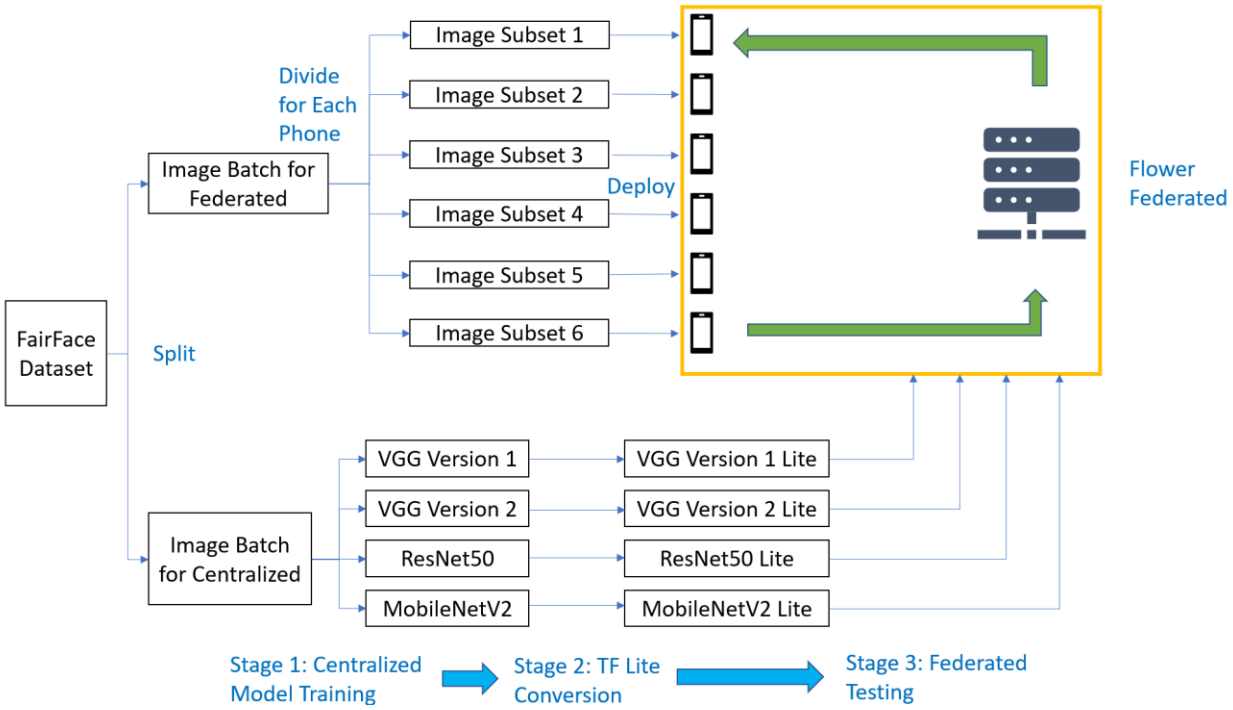


Figure 1. Project workflow overview.

This project consists of four major components demonstrated in Figure 1: preparing the input dataset, training CNN models (Stage One), converting them to TF Lite models (Stage Two), and testing them in Flower Federated (Stage Three). To preprocess the image dataset, we first divide it into two batches: federated and centralized. On the federated end, we further split the batch into even subsets that are each deployed on an Android device. On the centralized end, we use the entire image batch to train four model architectures in Stage One. Next in Stage Two, we convert each trained model to its TF Lite version to reduce the size. Finally, in Stage Three, we deploy each TF Lite model in the federated system to test its performance with the federated image batch.

3.2 DATASET

Most of the popular face datasets have a high percentage of Caucasian face images. This racial imbalance can create biases when the images are used to train ML models. Example dataset information can be found in Table 1 which is presented in [32, p. 5].

Table 1. Popular public face datasets that are more biased towards the Caucasian population [32, p. 5].

Dataset	Year	Number of Photos	Caucasian	Asian	Indians	African
CASIA-Web-FACE	2014	494K	84.5	2.6	1.6	11.3
VGGFace2	2017	3.31M	74.2	6.0	4.0	15.8
MS-Celeb-1M	2016	10M	76.3	6.6	2.6	14.5
LFW	2007	13K	69.9	13.2	2.9	14.0
IJB-A	2015	25K	66.0	9.8	7.2	17.0

In 2019, Kärkkäinen K. and Joo J. [27] published the FairFace dataset by collecting non-celebrity face images from YFCC-100M Flickr, which are labeled with age, gender, and race. This gives us more information about the demographic backgrounds of the dataset.

3.2.1 *Understanding the Content*

FairFace has a total of 97698 images with 7 labels: Black (12.5%), East Asian (12.6%), Indian (12.6%), Latino Hispanic (13.7%), Middle Eastern (9.4%), Southeast Asian (11.0%), and White (16.9%). With a similar percentage of images per race, FairFace shows a balanced distribution of seven racial backgrounds. The images are further evaluated using Resnet34 facial embedding that was pretrained on more biased face datasets. The result demonstrates that FairFace images are loosely connected and well spread in the 2D space, indicating a better racial coverage and diversity

[27]. These attributes can help train ML models with a better generalization performance in terms of fairness compared to the biased datasets in Table 1.

FairFace contains two sets of training and validation images with two padding levels as shown in Table 2. It comes with two CSV files that have information about the file name, age, gender, and race attributes for each image.

Table 2. Overview of the original image distribution in the dataset.

Dataset Name	Image Padding	Image Dimension	Number of Train Images	Number of Validation Images
FairFace	0.25	224 × 224	86744	10954
FairFace	1.25	448 × 448	86744	10954

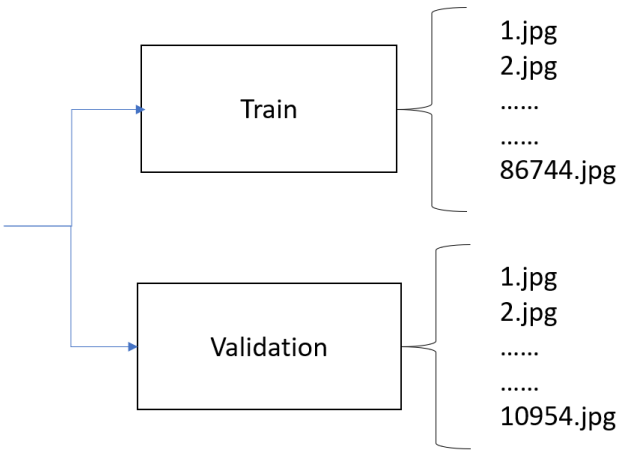
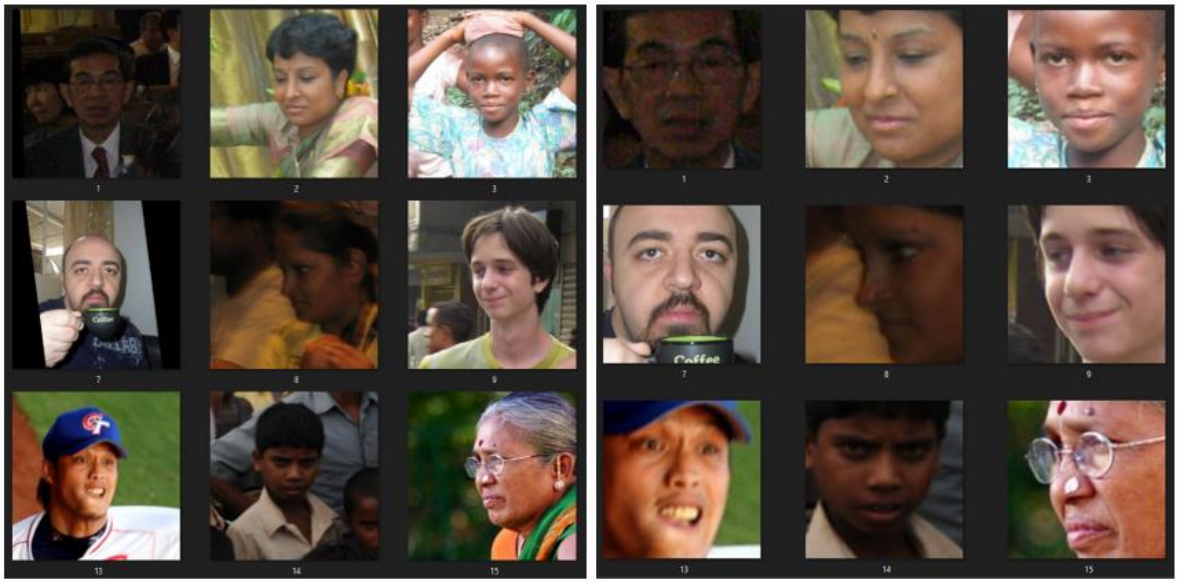


Figure 2. Original folder structure of the dataset that contains only two folders: train and validation.

Figure 2 shows the original folder structure. However, these images cannot be used for ML model training as they are not yet associated with their labels. We need to create a new structure that groups these images into different sub-folders to create the association.



Padding = 1.25 Padding = 0.25

Figure 3. A side-to-side comparison between the two different Padding values.

Figure 3 demonstrates how Padding values affect the images. For each image on the left with Padding = 1.25, there is a zoomed-in version on the right that focuses on the face only. The 0.25 Padding version was created by applying a face detection algorithm using the dlib library [27].

3.2.2 *Preprocessing*

We first choose to only use the image set with a 0.25 Padding value. This is mainly because each image has fewer background noises and more facial information. With a limited memory capacity, a smaller image size also allows Android devices to load more images during training. We then transform the dataset folder structure so each image can be associated with its label from the CSV files.

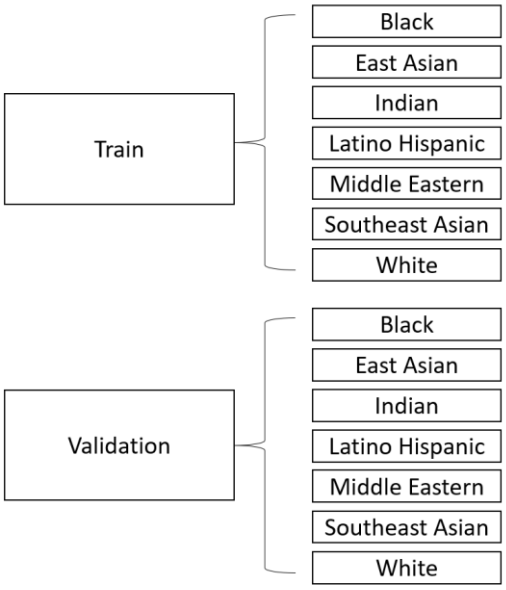


Figure 4. Preprocessed folder structure for the image dataset. Each race group contains only images with the same label.

The original image dataset is divided into 7 sub-folders based on the "race' attribute for training and validation, respectively. This new structure shown in Figure 4 meets the API input requirement of Keras ImageDataGenerator [33], which provides a convenient tool to load training data from directories.

Table 3. Overview of image distribution per race for the learning environments.

	Central Train	Central Validation	Federated Train	Federated Validation
Black	11233	1456	1000	100
East Asian	11287	1450	1000	100
Indian	11319	1416	1000	100
Latino Hispanic	12367	1523	1000	100
Middle Eastern	8216	1109	1000	100

Southeast Asian	9795	1315	1000	100
White	15527	1985	1000	100
Total	79744	10254	7000	700

Next, we create two sets of images with the same structure shown in Figure 4 for centralized and federated training, respectively. The majority of the images will be trained in a centralized setting to see how well the models can learn. A small portion of the images will be split for federated learning on Android. This is so that the models trained in the centralized environment will have new images to learn information from in the federated framework. A more detailed overview of the image assignment can be found in Table 3.

3.2.3 *Practical Challenges with Racial Classification*

Kärkkäinen K. and Joo J. [27] tested the cross-dataset classification accuracy based on the race attribute when they published FairFace. They noted that their model has a 93.7% accuracy on White Race, and a 75.4% accuracy on Non-White Races including four race categories: White, Black, Asian, and Indian. Even though the dataset contains fewer racial biases than most other face datasets, it also shows that the accuracy decreases when more race categories are used for training. This is a difficult task with practical challenges to identify a given ethnicity, as people of the same ethnicity can look drastically different, and people of different ethnicities can look very similar. Our project uses seven race categories, and it becomes even harder to make the classification. It is expected that our model accuracy should be lower than 75.4%.

3.3 STAGE 1: PREPARING CNN MODELS WITH CENTRALIZED TRAINING

We prepared and trained four CNN model architectures in the centralized environment: VGGFace

[34] Version 1, VGGFace Version 2, ResNet50 [26], and MobileNetV2 [35]. By experimenting with different parameter settings and applying various training techniques, we hope to learn how well each model learns from the centralized image batch and to evaluate which is the most suitable to test in the federated system. For VGGFace, we loaded pre-trained facial weights and used transfer learning with two model architecture versions. For ResNet50 and MobileNetV2, we randomly initialized model weights and trained the entire model. The main goal of Stage 1 is to have the model absorb as much knowledge as possible in the centralized environment.

3.3.1 *VGGFace Version 1*

The VGGFace model is a deep CNN model that has been trained on the Visual Geometry Group [34] (VGG) dataset of over two million face images. The pretrained weights have learned generic patterns of facial features under the model structure. We utilize transfer learning to reuse such high-level knowledge and teach the model more low-level knowledge about FairFace. The key to doing so is to freeze some base layers and only train a certain number of layers on top.

The first architecture version we propose is to add a head model with new layers on top of the base VGGFace model and only train the head with FairFace. The structure shown in Table 4 has the first 19 layers (including Input, MaxPooling2D, and Flatten) as base layers, where all the trainable parameters are frozen. Layers 20-27 (marked in bold) are the new head layers and are trained with FairFace data.

Table 4. VGGFace model architecture with new head layers.

Layer Number	Layer Function	Output Shape	Number of Parameters	Trainable
1 (Input)	InputLayer	[(None, 224, 224, 3)]	0	No

2	Conv2D	(None, 224, 224, 64)	1792	No
3	Conv2D	(None, 224, 224, 64)	36928	No
4	MaxPooling2D	(None, 112, 112, 64)	0	No
5	Conv2D	(None, 112, 112, 128)	73856	No
6	Conv2D	(None, 112, 112, 128)	147584	No
7	MaxPooling2D	(None, 56, 56, 128)	0	No
8	Conv2D	(None, 56, 56, 256)	295168	No
9	Conv2D	(None, 56, 56, 256)	590080	No
10	Conv2D	(None, 56, 56, 256)	590080	No
11	MaxPooling2D	(None, 28, 28, 256)	0	No
12	Conv2D	(None, 28, 28, 512)	1180160	No
13	Conv2D	(None, 28, 28, 512)	2359808	No
14	Conv2D	(None, 28, 28, 512)	2359808	No
15	MaxPooling2D	(None, 14, 14, 512)	0	No
16	Conv2D	(None, 14, 14, 512)	2359808	No
17	Conv2D	(None, 14, 14, 512)	2359808	No
18	Conv2D	(None, 14, 14, 512)	2359808	No
19	MaxPooling2D	(None, 7, 7, 512)	0	No
20	**Flatten**	(None, 25088)	0	Yes
21	**Dense**	(None, 512)	12845568	Yes
22	**BatchNormalization**	(None, 512)	2048	Yes
23	**Dropout**	(None, 512)	0	Yes
24	**Dense**	(None, 256)	131328	Yes

25	**BatchNormalization**	(None, 256)	1024	Yes
26	**Dropout**	(None, 256)	0	Yes
27 (Predictions)	**Dense**	(None, 7)	1799	Yes

The original VGGFace model has 14,714,688 trainable parameters, a lot of which are in the lower base layers and have high-level knowledge about faces. We use this knowledge to extract facial features from FairFace and *Flatten* them to a 1-D array through the 20th layer. Each neuron in Layers 21, 24, and 27 is connected to all the neurons from the previous layer. The purpose of a *Dense* layer is to use an activation function to convert an input to an output. *BatchNormalization* and *Dropout* layers are commonly used in combination to reduce the overfitting effect.

Table 5. VGGFace and ResNet50 compile and training hyperparameters.

Input Dimension	224 × 224 × 3
Batch Size	32
Optimizer	Adam
Learning Rate	0.001
Loss Function	Categorical Cross Entropy
Number of Epochs	20
EarlyStopping Patience	3

Table 5 shows the model compile and training hyperparameters. Using a Patience value for *EarlyStopping*, training terminates if the performance does not improve over that number of epochs. Altogether with a small *Learning Rate* and a large *Number of Epochs*, this mechanism can reduce the effect of improper model training.

3.3.2 *VGGFace Version 2*

We constructed the second version of VGGFace by freezing a portion of the bottom layers and only training the top few layers without adding a complex head model. When calling the VGGFace class constructor, if *include_top* is set to *True*, the model expects an output of 2622 classes. To classify 7 classes, we need to first set *include_top* to *False* to remove the prediction layer. We can then add one *GlobalAveragePooling2D* layer followed by a *Dense* layer with 7 neurons representing 7 class outputs.

Table 6. VGGFace model architecture without new head layers.

Layer Number	Layer Function	Output Shape	Number of Parameters	Trainable
1 (Input)	InputLayer	[(None, 224, 224, 3)]	0	No
2	Conv2D	(None, 224, 224, 64)	1792	No
3	Conv2D	(None, 224, 224, 64)	36928	No
4	MaxPooling2D	(None, 112, 112, 64)	0	No
5	Conv2D	(None, 112, 112, 128)	73856	No
6	Conv2D	(None, 112, 112, 128)	147584	No
7	MaxPooling2D	(None, 56, 56, 128)	0	No
8	Conv2D	(None, 56, 56, 256)	295168	No
9	Conv2D	(None, 56, 56, 256)	590080	No
10	Conv2D	(None, 56, 56, 256)	590080	No
11	MaxPooling2D	(None, 28, 28, 256)	0	No
12	Conv2D	(None, 28, 28, 512)	1180160	No
13	Conv2D	(None, 28, 28, 512)	2359808	No

14	Conv2D	(None, 28, 28, 512)	2359808	No
15	MaxPooling2D	(None, 14, 14, 512)	0	No
16	Conv2D	(None, 14, 14, 512)	2359808	Yes
17	Conv2D	(None, 14, 14, 512)	2359808	Yes
18	Conv2D	(None, 14, 14, 512)	2359808	Yes
19	MaxPooling2D	(None, 7, 7, 512)	0	Yes
20	**GlobalAveragePooling2D**	(None, 512)	0	Yes
21 (Predictions)	**Dense**	(None, 7)	3591	Yes

Table 6 shows this model architecture where only Layer 20 and 21 (marked in bold) are added to the base model. Now the second version of VGGFace has a fully constructed architecture. To reuse the general knowledge in the bottom layers, we freeze the base layers and only enable training in Layer 16 to Layer 21 using FairFace.

This variation of the VGGFace model has most of the layers pretrained on the VGG Face Database. By retraining only the top 6 layers, the model can adapt to refined low-level facial knowledge that is specific to FairFace and adjust the weights accordingly. Most of the weight correction takes place in Layer 16 to Layer 18 where the majority of the 7,083,015 trainable parameters are. We use the same compile and training hyperparameters for this architecture as VGGFace Version 1, which can be found in Table 5.

3.3.3 *ResNet50*

ResNet50 is another popular deep CNN model. It has 177 layers, where 107 of them contain trainable parameters of 23,548,935 in total. The Keras library provides two weight choices. If

imported with *ImageNet* [18], the model loads weights that are pretrained from the ImageNet database to identify 1000 classes of common objects. If imported with *none*, the weights will be randomly initialized. As the 1000 classes in ImageNet do not include many examples of human faces, loading ImageNet weights does not help much with our Facial Recognition task. Transfer Learning is also not applicable as the base model does not have relevant knowledge regarding faces. Thus, we included all of the trainable parameters in the training process for ResNet50. The compile and training hyperparameters can be found in Table 5.

3.3.4 *MobileNetV2*

In comparison to the other three models mentioned, MobileNetV2 [35] is the lightest with only 2,232,839 trainable parameters by default. This compact model architecture uses much less memory to load in a computing environment than VGGFace and ResNet50. This feature allows MobileNetV2 to effectively run on devices with limited hardware resources without sacrificing too much performance in the meantime.

In the Keras library, MobileNetV2 has the option to load weights trained on ImageNet. As we previously discussed in Section 3.3.3, using ImageNet weights for transfer learning is also not applicable to this project. Thus, the best solution is to randomly initialize the entire model weights and use FairFace to train every parameter from scratch.

Table 7. MobileNetV2 model compile settings and training hyperparameters.

Input Dimension	224 × 224 × 3
Batch Size	32
Optimizer	SGD
Learning Rate	0.001

Momentum	0.9
Loss Function	Categorical Cross Entropy
Number of epochs	50
EarlyStopping Patience	3

Compared to the Table 5 settings for VGGFace and ResNet50, it is noticeable that the optimizer for MobileNetV2 is different as shown in Tabel 7. Although the Adam optimizer generally converges faster, we wanted to use a different optimizer such as Stochastic Gradient Descent (SGD) [36] that generalizes better to test the performance impact.

3.4 STAGE 2: CONVERTING TRAINED MODELS TO TENSORFLOW LITE

The TensorFlow CNN models that we prepared in Stage 1 are generally too heavy for mobile devices, embedded Linux, and microcontrollers, due to the limitations of memory and computing power. TensorFlow Lite (TF Lite) provides a solution for on-device model personalization without losing much of the knowledge learned in Stage 1. The main goal for Stage 2 is to reduce the CNN models we prepared in Stage 1, so we can deploy them in Stage 3.

3.4.1 *TF Lite Optimization*

One of the biggest optimizations is the reduction of the model size. After a TF model is converted into a TF Lite model, the storage size requirement is lowered. The Android application that contains the model installation will also be more efficient. Furthermore, when such models are loaded in the training process, the memory usage is smaller, and the model inference latency is reduced. Typically, this optimization comes with a trade-off of changes in model accuracy. While

it is more common for the model to lose some accuracy, some of our models gained a small amount of accuracy.

3.4.2 *Conversion Process*

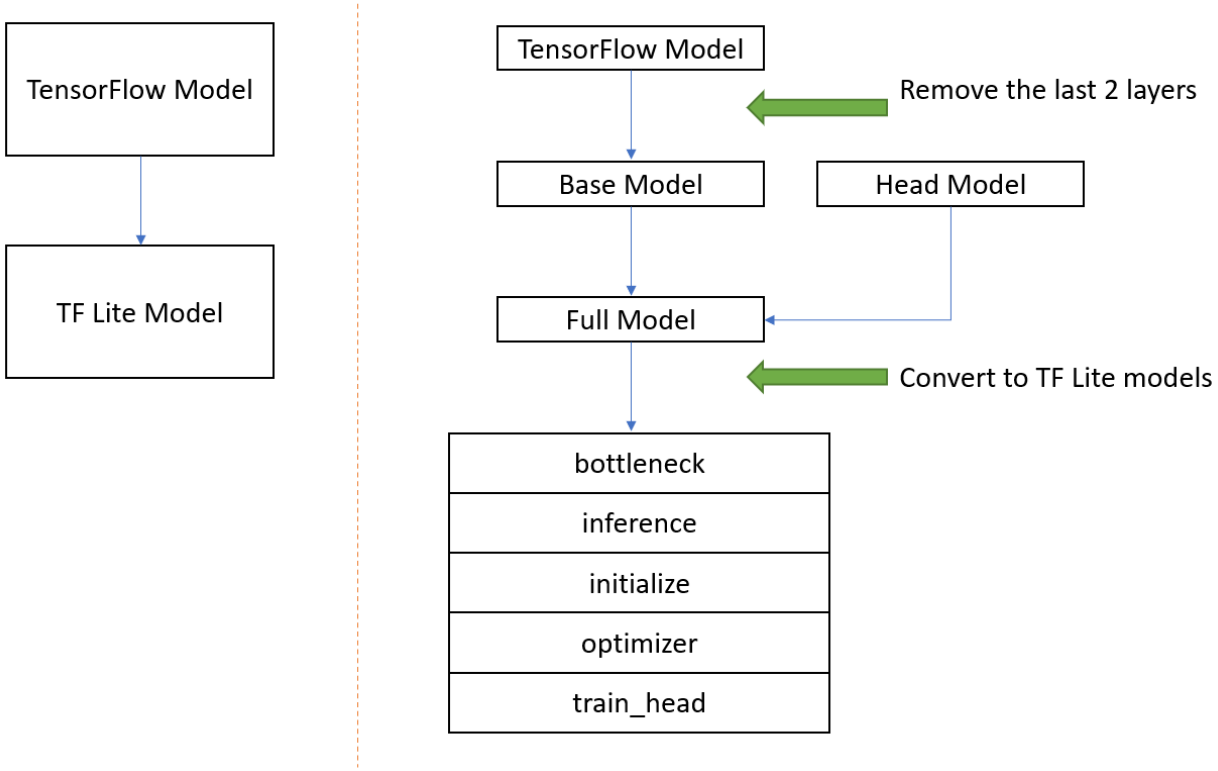
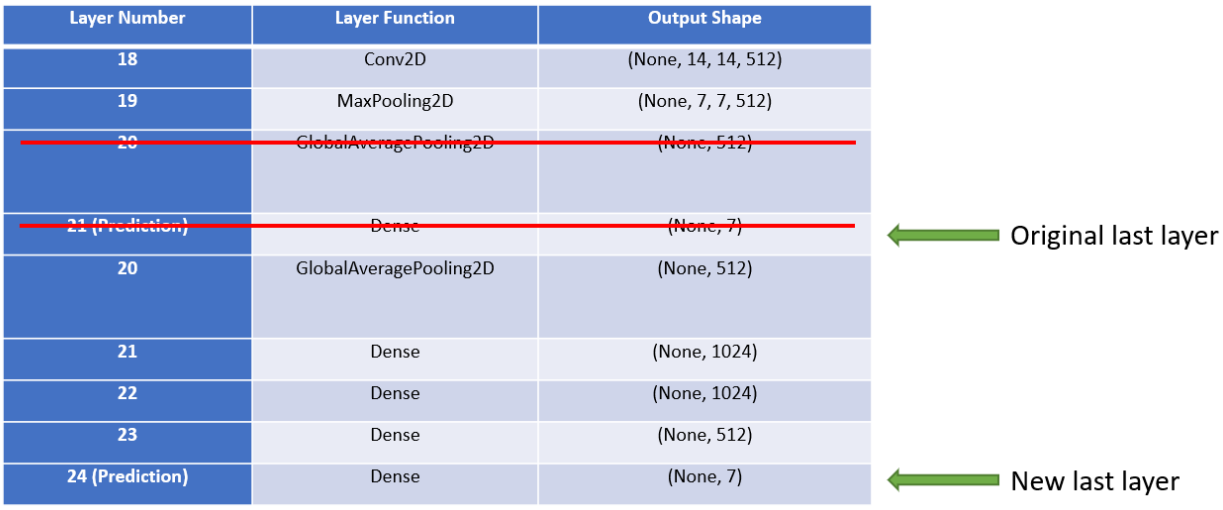


Figure 5. Comparison between the official *TF Lite* library (Left) and Flower *tfltransfer* library (Right).

The latest official TF Lite library does a one-to-one conversion. It takes a TensorFlow model as the input, which is then converted into one multi-signature *tflite* model. On the other hand, the Flower framework used in our project provides a *tfltransfer* library, which takes two TensorFlow models as the input and produces five *tflite* models with different signatures. Figure 5 demonstrates the differences between the conversion processes in more detail. As model personalization is

essentially a form of on-device transfer learning, we set up a Base Model and a Head Model as the input of *tfltransfer*.



Layer Number	Layer Function	Output Shape	
18	Conv2D	(None, 14, 14, 512)	
19	MaxPooling2D	(None, 7, 7, 512)	
~~20~~	~~GlobalAveragePooling2D~~	~~(None, 512)~~	
~~21 (Prediction)~~	~~Dense~~	~~(None, 7)~~	Original last layer
20	GlobalAveragePooling2D	(None, 512)	
21	Dense	(None, 1024)	
22	Dense	(None, 1024)	
23	Dense	(None, 512)	
24 (Prediction)	Dense	(None, 7)	New last layer

Figure 6. Illustration of layer manipulation to combine Base Model with Head Model for Resnet50.

The Base Model, if including the top, has a *prediction* layer as the final layer, and often a *GlobalAveragePooling2D* layer before that. To construct a Full Model, we need to first remove the last 2 layers from the Base Model and add a Head Model, where the input of the first layer matches the dimension of the last output layer in the Base Model. Figure 6 demonstrates how all four model architectures trained in Stage 1 are modified to meet the requirements of the *TFLiteTransferConverter* function from the *tfltransfer* library in Stage 2.

To start the conversion process, we also need to specify a few arguments, such as model *optimizer function* (and *learning rate* if the optimizer is SGD), *batch size*, and *number of classes*. Using the same settings as those in Table 5 and Table 7 for the Base Model centralized learning parameters, we can convert the four CNN models to their TF Lite versions as a result.

3.5 Stage 3: Deploying TF Lite Models to Federated Learning

In this stage, we aim to test if the four TF Lite models converted in Stage 2 have a size small enough for Android devices, and how well they can perform in the Flower FL environment. The main goal of Stage 3 is to see if the previously trained models can continue learning in the federated environment after the conversion.

3.5.1 *Flower Backend*

Flower is a Federated Learning framework that orchestrates the main components of this stage. By setting up a server and Android clients, the system separates the sensitive data from the central server and provides a private way for users to participate in training. Flower connects the server and the clients by matching the same IP address and port number. gRPC [35] is the main communication protocol to transfer the model updates and complete weight aggregation.

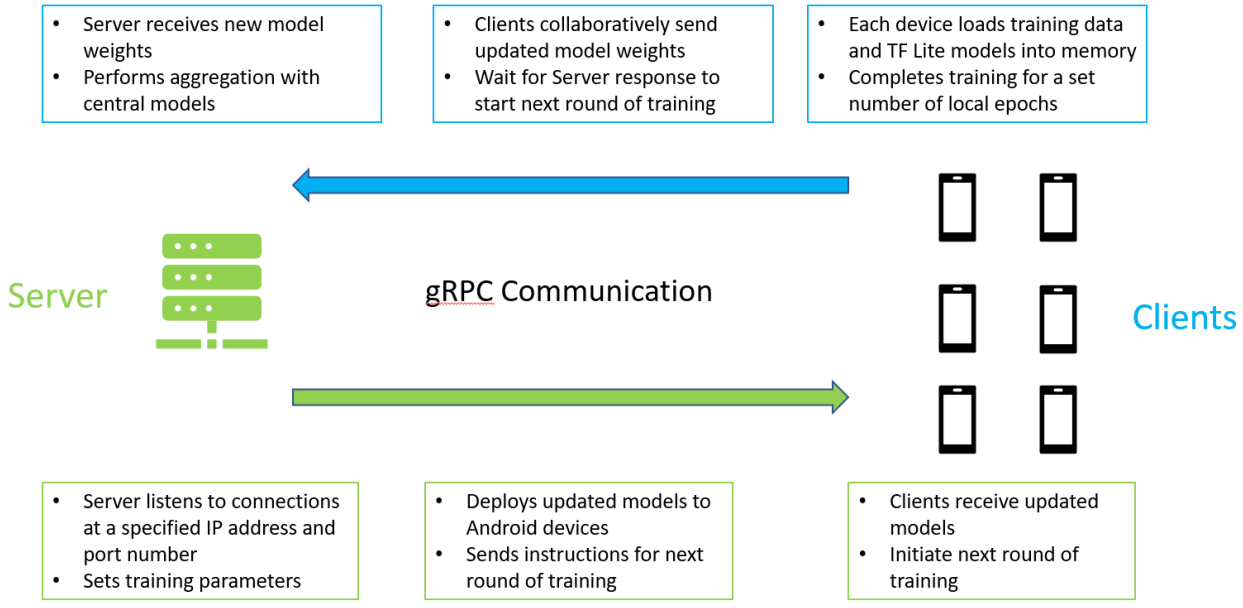


Figure 7. System Architecture of Flower Android framework.

Figure 7 shows the overall system operation flow of the FL process. Initially, the server is started by listening to a specific IP address with a port number. Once there are enough devices establishing connections with the server, training takes place locally on each device for a specific number of epochs. After each round of training, all the devices send the updates to the server for model weight aggregation. With the new knowledge gained, the server deploys the latest models back to each device and the cycle continues until the number of global rounds is complete.

3.5.2 *Server*

As model aggregation is CPU intensive, the server is hosted on a desktop with a powerful CPU for faster computation. The server script specifies the IP address, port number, fitting parameters, and aggregation strategy. A summary of the server hardware and software specifications can be found in Table 8.

Table 8. Server Specification.

CPU	GPU	Memory Size	Disk Storage	Python Version	Core Packages	Local Epochs	Global Rounds
AMD R7 5800X	NVIDIA RTX 3090	32GB DDR4	2TB SSD	3.7.10	tflite=2.3.0 tensorflow=2.7.0 keras=2.7.0 flwr=0.18.0	20	20

FedAvgAndroid is the only aggregation algorithm available in Flower 0.18.0 for Android devices. The implementation is based on the popular *FedAvg* algorithm proposed by McMahan et al. [36]. Since Android applications are in binary format, *FedAvgAndroid* adds custom serialization and deserialization during the server and client communication.

3.5.3 *Client*

In our experiments, we used Android Studio to simulate virtual Android clients. All the emulators are created with the same system specifications shown in Table 9. We discovered that the testing environment can have a maximum of 6 simulated devices running concurrently before reaching the memory limit.

Table 9. Android Device Specification.

Model	Release Name	API Level	Application Binary Interface	Target
Google Pixel 4	R	30	x86	Android 11.0 (Google Play)

Although we previously allocated 7000 training images and 700 test images for federated testing from the original dataset, we noticed loading more than 400 images at once would crash the Android application. As a result, we decided to use 350 training images and 35 testing images per device consistently across the 6 devices, making the data Independent and Identically Distributed (IID).

Figure 8 demonstrates how the user chooses which subset to load into memory by entering a number between 1 and 20. Each subset has 50 different training images and 5 testing images per race. A total of 385 images and the TF Lite models we prepared in Stage 2 were loaded into memory before the training begins.

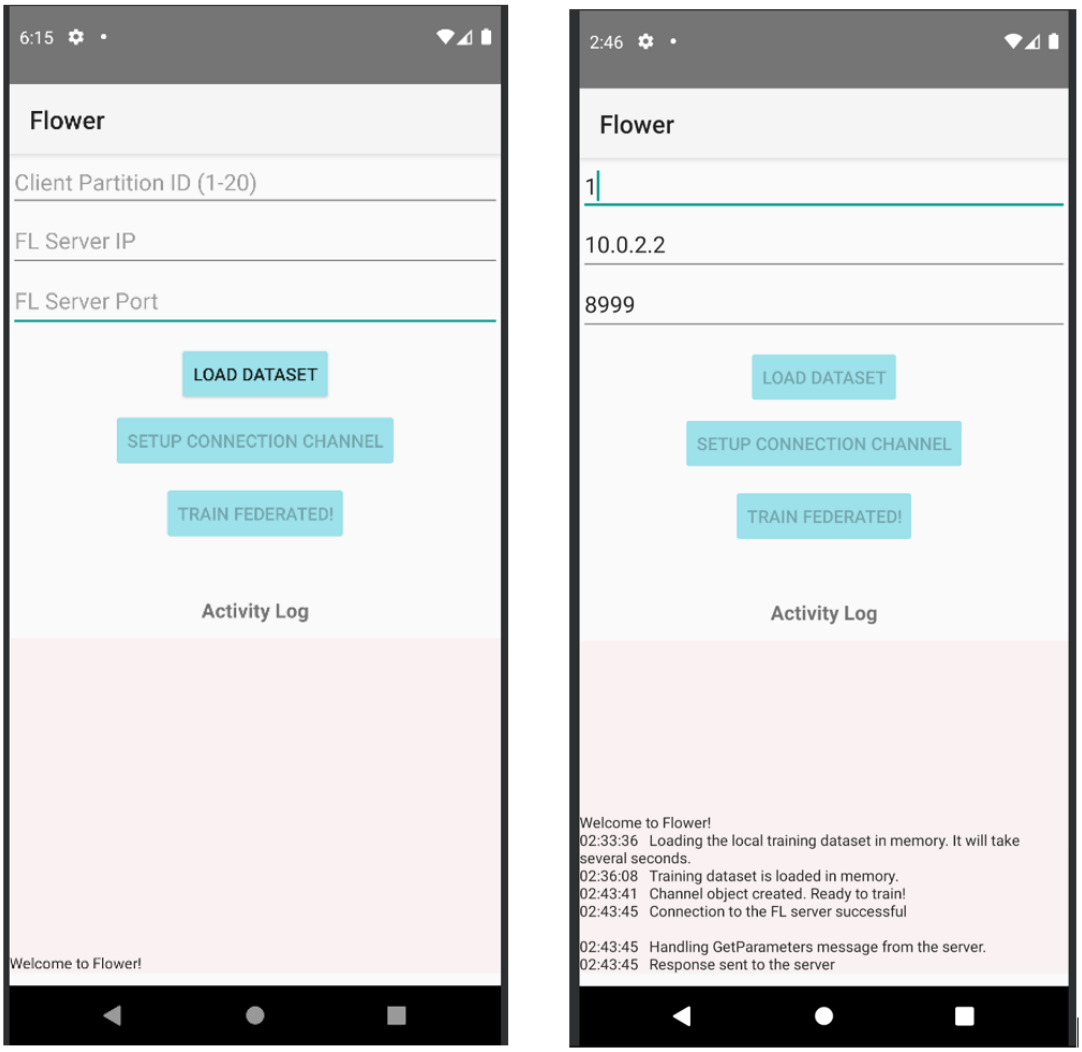


Figure 8. Flower Android Application GUI.

3.5.4 *Communication*

Flower uses gRPC for the bidirectional data communication between the server and clients. gRPC is a high-performance Remote Procedure Call (RPC) that can efficiently transmit the model weights. In each of the training rounds as new Android client devices connect, the private data stays on the user devices with the TF Lite models. At the end of the training, the newly updated weights will be transferred to the server via gRPC. By staying in the same channel, the server

receives new model weights from edge devices and performs aggregation. The newly updated TF Lite models are deployed back to each edge device to prepare for the next round of training. User privacy with sensitive photos is guaranteed by only sharing the model weights with the server.

3.6 Evaluation Metrics

Throughout the three stages of training, converting, and testing, we consider various evaluation metrics to measure the success of this project. In Stage 1, we measure how heavy CNN models are and how well they can learn. These two factors are commonly a trade-off pair, which is dominated by the total number of trainable parameters and the final accuracy the model can reach. Ideally, for the model to deploy on an Android device, it should have a small number of parameters while maintaining a decent accuracy. Next in Stage 2, we measure how much the *tfliteconverter* function can reduce the original model complexity. The final TF Lite model size is a crucial indicator of whether it is a viable option to test the federated system. Finally in Stage 3, we want to evaluate how well the TF Lite models can perform in Flower. We examine the test accuracy convergence throughout the training process and measure the device resource usage on Android in terms of CPU, memory, and energy.

Chapter 4. EXPERIMENT RESULTS AND EVALUATION

We examined various results regarding the four models throughout the three main stages of this project. At the end of each stage, we analyzed and compared the performance differences between the models. The specific attributes that were measured in the three stages are shown as the following:

1. Model accuracy, loss, and the number of parameters
2. Size reduction from the conversion
3. Model test accuracy and Android resource usage in Flower

4.1 STAGE 1: CNN MODEL TRAINING

To best evaluate model performance, we focus on accuracy, loss, and the number of parameters for the four CNN models in the centralized environment. During each training or validation round, accuracy is the percentage of correct predictions that the model makes, and loss is the summation of the errors in that process. These two metrics combined can be used to determine if a model is learning properly and correctly. Meanwhile, the total number of parameters is a dominant indicator of how heavy a model is.

4.1.1 *VGGFace Version 1*

VGGFace Version 1 was constructed by adding a head model on top of the base model. We used transfer learning to load the face weights in the base and train only the head layers. This architecture contains 27,696,455 total parameters and 12,980,231 trainable parameters.

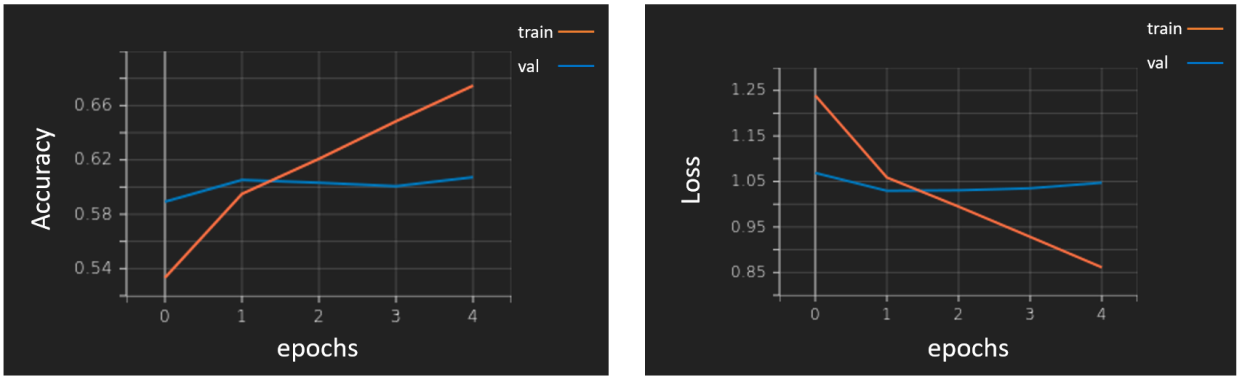


Figure 9. VGGFace Version 1 training and validation convergence plots.

The model quickly learned new facial information using knowledge from the pre-trained weights, as Figure 9 demonstrates validation accuracy increased to 58.92% after the first epoch. However, it struggled to learn further from FairFace data and training finished after 5 epochs with *EarlyStopping*. The left plot shows in the last 3 epochs validation accuracy stopped growing, and training accuracy increased to 67.46%. This indicates that the model did not effectively learn during the period, which can be also verified by the increase in loss value in the right plot. The loss value went from 1.0295 to 1.0473. A high loss value shows that the model was making more prediction errors. The peak model validation accuracy throughout the training process is 60.73%.

Additionally, we used a few methods in an attempt to increase the peak validation accuracy, such as data augmentation and fine-tuning the layer architecture. For data augmentation, we applied rotation, translation, and cropping to the original FairFace images to produce additional input. In theory, this can increase the input complexity and provide more information for the model to adapt to. For layer architecture, we changed the number of neurons in the Dense layer, the ratio of the Dropout layer, and the number of Dense layers in the head model. However, none of the methods showed a significantly positive impact on the model performance.

4.1.2 *VGGFace Version 2*

Next, we explored the training for VGGFace Version 2, where we froze the first 15 layers and only trained Layer 16-21. VGGFace Version 2 has 14,718,279 total parameters and 7,083,015 trainable parameters.

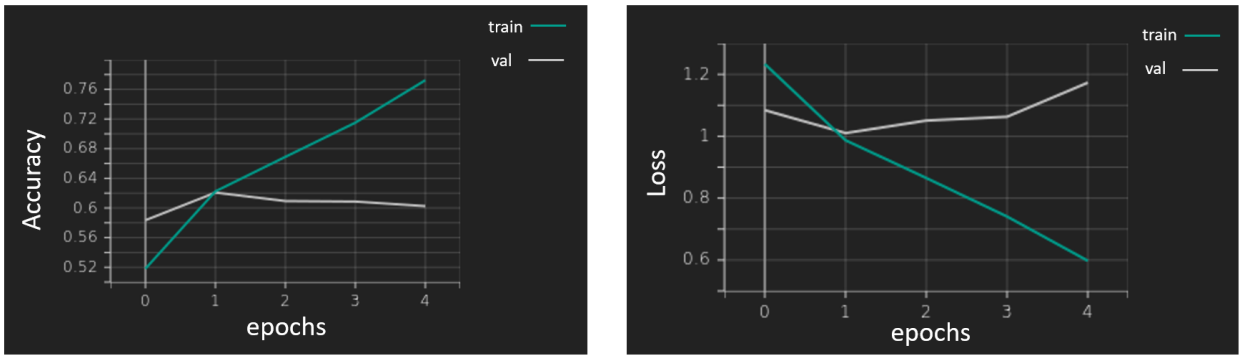


Figure 10. VGGFace partially freezing training and validation convergence plots.

We used the same training hyperparameters to be consistent with VGGFace Version 1. Figure 10 shows after the first epoch, the model reached a validation accuracy of 58.35% due to the pretrained weights. Similar to VGGFace Version 1, this model was unable to obtain new knowledge after the first epoch and had an increasing loss at the same time. The validation accuracy was at a peak with 62.07% in the second epoch, but it started to decline afterward. During this period, the right plot shows loss increased from 1.0102 to 1.1732. The training terminated early after 5 rounds because of our *EarlyStopping* mechanism.

4.1.3 *ResNet50*

ResNet50 by default is a deep CNN model heavier than VGGFace. With 7 classes it has 23,602,055 total parameters and 23,548,935 trainable parameters. We trained the entire model from scratch without loading any pre-trained weights.

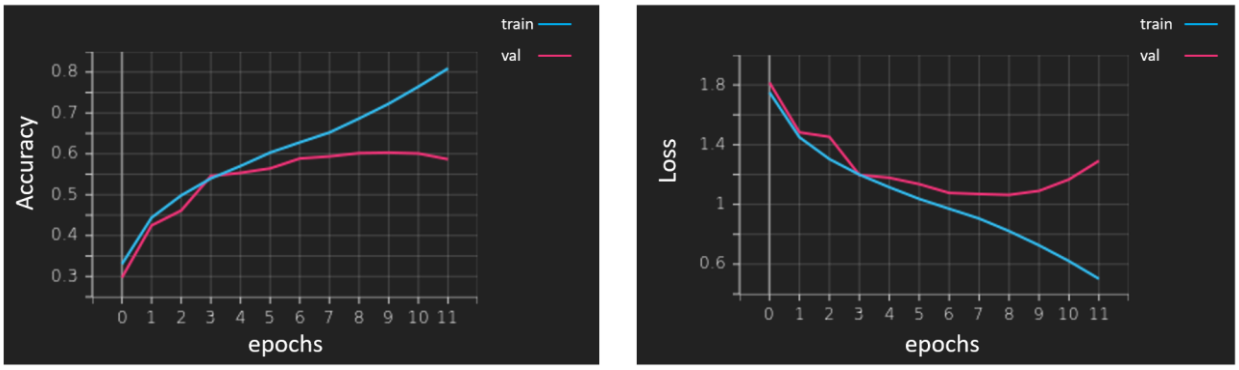


Figure 11. ResNet50 training and validation convergence plots.

We used the same training hyperparameters as VGGFace. Figure 11 shows that after the first epoch, ResNet50 reached a validation accuracy of 33.01%. This is much lower than the VGGFace models as ResNet50 model weights were randomly initialized. The left plot shows a longer and more consistent increase in accuracy. However, after the eighth epoch, the model started to struggle with learning new information from FairFace. It can be confirmed when loss increased from 1.0906 to 1.2920 shown in the right plot until the training was stopped by *EarlyStopping*. The peak validation accuracy is 60.29%.

4.1.4 *MobileNetV2*

MobileNetV2 by default is much lighter than VGGFace and ResNet50. With 7 classes, it has 2,266,951 total parameters and 2,232,839 trainable parameters. This attribute allows faster training and requires less memory usage in the process. However, it is generally expected that the accuracy of the model can be lower than that of a heavier model.

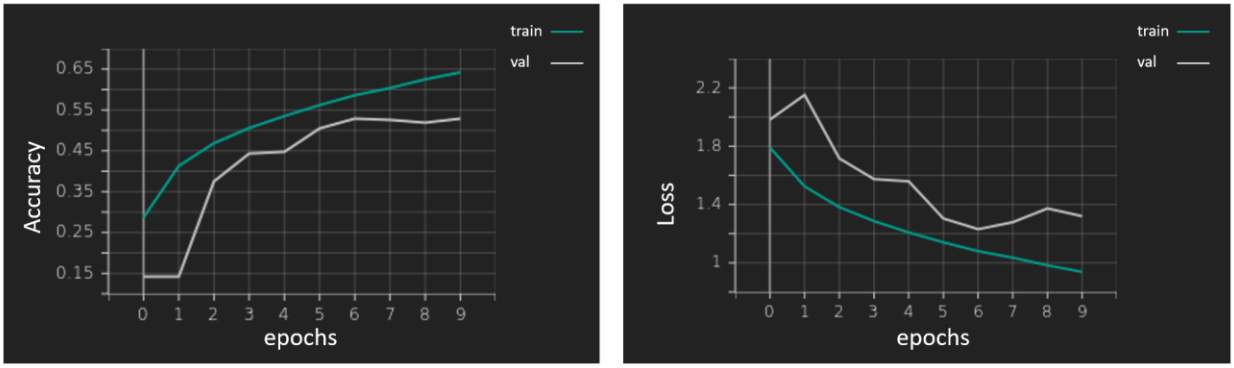


Figure 12. MobileNetV2 training and validation convergence plots.

We used all trainable parameters without loading any pre-trained weights. Figure 12 shows MobileNetV2 only reached a 28.44% validation accuracy after the first epoch. It is lower than all of the other models we tested so far. The left plot shows the model had a consistently increasing learning curve until loss increased from 1.2725 to 1.4581 in the last 3 epochs and led to an early stop. This model has a peak validation accuracy of 52.88%.

4.1.5 *Model Comparison and Analysis*

Table 10. CNN model comparison.

	VGGFace Version 1	VGGFace Version 2	ResNet50	MobileNetV2
Peak Accuracy	60.73%.	**62.07%**	60.29%	52.88%
Loss	1.0473	1.0102	1.0906	1.2725
Total Parameters	27,696,455	14,718,279	23,602,055	**2,266,951**
Trainable Parameters	12,980,231	7,083,015	23,548,935	2,232,839

Table 10 shows that VGGFace Version 2 has the highest validation accuracy of 62.07% and MobileNetV2 has the lowest validation accuracy of 52.88%. However, to increase the accuracy to above 60%, the model has to increase 7 to 10 times in size based on the total number of parameters.

To deploy a functional model with good performance on Android, we need to carefully examine the model size after the Stage 2 conversion and evaluate which is the best candidate.

4.2 STAGE 2: TF LITE CONVERSION

We performed TF Lite conversion for all four models that we trained in Stage 1. We measured the original mode size, the final model size, and the size of each TF Lite file after the conversion.

Table 11. Overview of model sizes before and after the conversion.

Model	VGGFace Version 1	VGGFace Version 2	ResNet50	MobileNetV2
Original Size	205 MB	110 MB	274 MB	20.9 MB
Final Size	64.1 MB	64.1 MB	103 MB	**19.4 MB**
Reduction Rate	68.7%	41.7%	62.4%	7.2%
bottleneck.tflite Size	57473 KB	57479 KB	91763 KB	8651 KB
inference.tflite Size	4 KB	4 KB	4 KB	4 KB
initialize.tflite Size	8213 KB	8213 KB	14357 KB	11285 KB
optimizer.tflite Size	16 KB	16 KB	16 KB	4 KB
train_head.tflite Size	12 KB	12 KB	12 KB	11 KB

Table 10 shows the Original Size is proportional to the number of total parameters each model has. VGGFace Version 1 had the highest Reduction Rate of 68.7% using TF Lite Converter, whereas MobileNetV2 only had a 7.2% Reduction Rate.

Various factors can determine the size of each of the five TF Lite component files after the conversion. VGGFace and ResNet50 produced a larger *bottleneck.tflite* model compared to MobileNetV2. The *Adam* optimizer used for VGGFace and ResNet50 caused *optimizer.tflite* to be slightly larger than that of MobileNetV2. Since we added the same head layers, the *train_head.tflite* files for these models are about the same size after the conversion.

It is worth noting that the Final Size of MobileNetV2 is only 19.4MB, significantly smaller than any of the other models. It is the most lightweight candidate for Android deployment, as large models can overload the device memory limit, negatively affect the training results, and sometimes cause the application to crash.

4.3 Stage 3: Flower Federated Training

We tested VGGFace (Version 1), ResNet50, and MobileNetV2 in this stage. VGGFace Version 2 was not selected because we expect it to behave like VGGFace Version 1, as the peak accuracy is similar and the Final Size is the same. For each of the three candidates, we measured test accuracy, test loss, and resource usage across six Android emulators. Figure 13 demonstrates the Android application GUI when models and images are both loaded in memory during the Flower training process.

Figure 13. Flower Android app GUI during training.

4.3.1 *VGGFace*

VGGFace TF Lite models have a Final Size of 64.1 MB as shown in Table 11. After loading the models into memory, the application GUI rendering became slow. Each device used less than 3 minutes to load 350 training and 35 test images. Training completed across 6 devices after 1067.5 seconds.

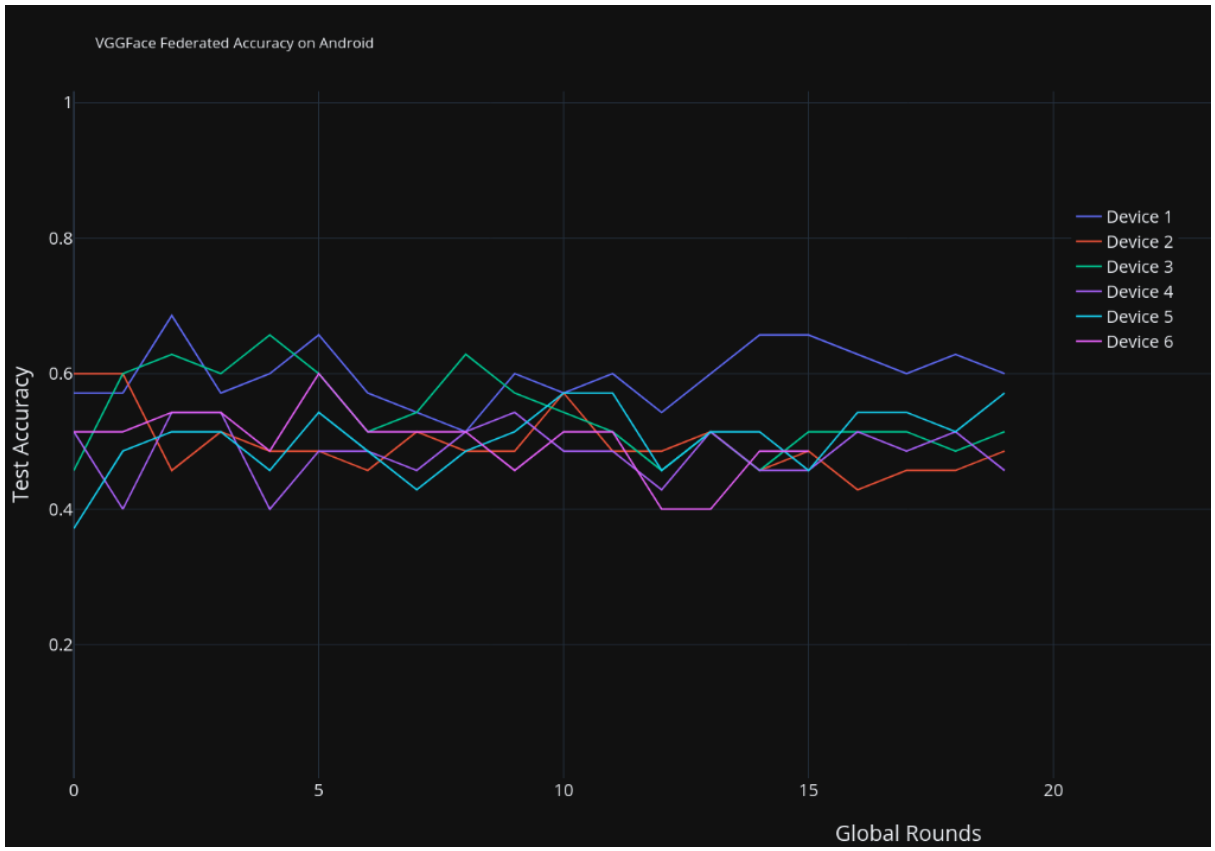


Figure 14. VGGFace Test Accuracy plot.

Figure 14 shows the changes in Test Accuracy over Global Rounds. The initial average Test Accuracy was 50.48% from 6 devices. After 20 global rounds of training with 20 local epochs for each device, the final average Test Accuracy increased slightly to 51.90%. However, Figure 14 shows the Test Accuracy on each device went up and down interchangeably and indicates the learning was unstable.

Figure 15. VGGFace Loss plot.

Although there is an increase in Test Accuracy, Figure 15 shows that loss increased significantly throughout the training process until it reached “Infinity”. A larger loss value means the model is making more prediction errors, which confirms the unstable learning we observed in Figure 14 previously.

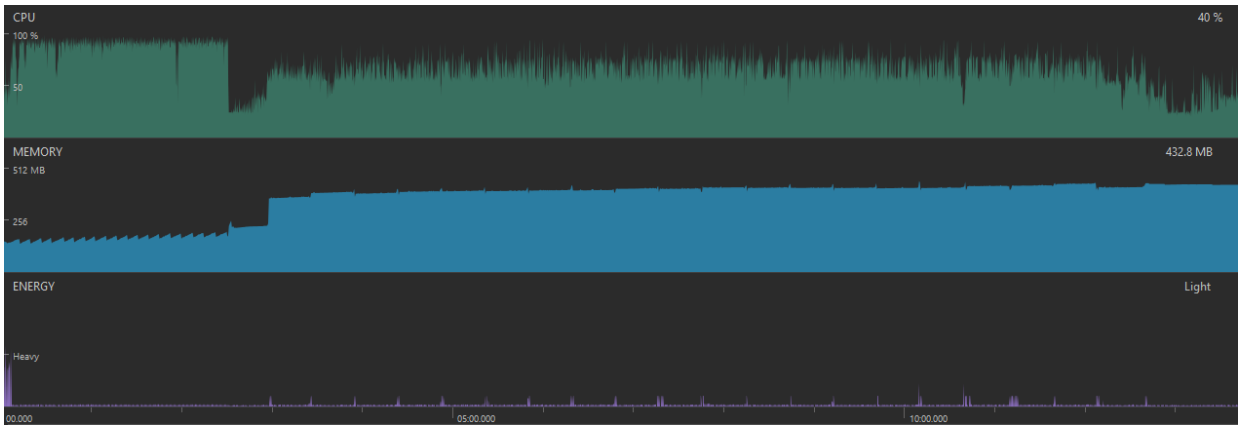


Figure 16. Android resource usage during VGGFace training.

We used Device Monitoring in Android Studio to record the resource usage of CPU, Memory, and Energy as shown in Figure 16. In the beginning, when loading the models and images, the CPU was constantly running near 100% and the Energy usage was heavy, which potentially caused the slow rendering of the application. In the training phase, the CPU usage was about 65% and memory usage was about 430 MB, while energy consumption stayed relatively low.

4.3.2 *ResNet50*

ResNet50 TF Lite models have a Final Size of 103 MB according to Table 11. Loading such a heavy model used a lot of memory on Android. We attempted to continue loading the images into memory, however, the application crashed each time due to memory restriction. We are unable to move on to the training phase to test the model performance and produce any results. This proves that ResNet50 is too heavy for Android on-device training even after being converted to TF Lite models.

4.3.3 *MobileNetV2*

MobileNetV2 has a Final Size of 19.4 MB after TF Lite conversion, which is 3 to 5 times smaller than VGGFace and ResNet50. Loading the models and images took about 35 seconds and the total training time was 723.2 seconds.

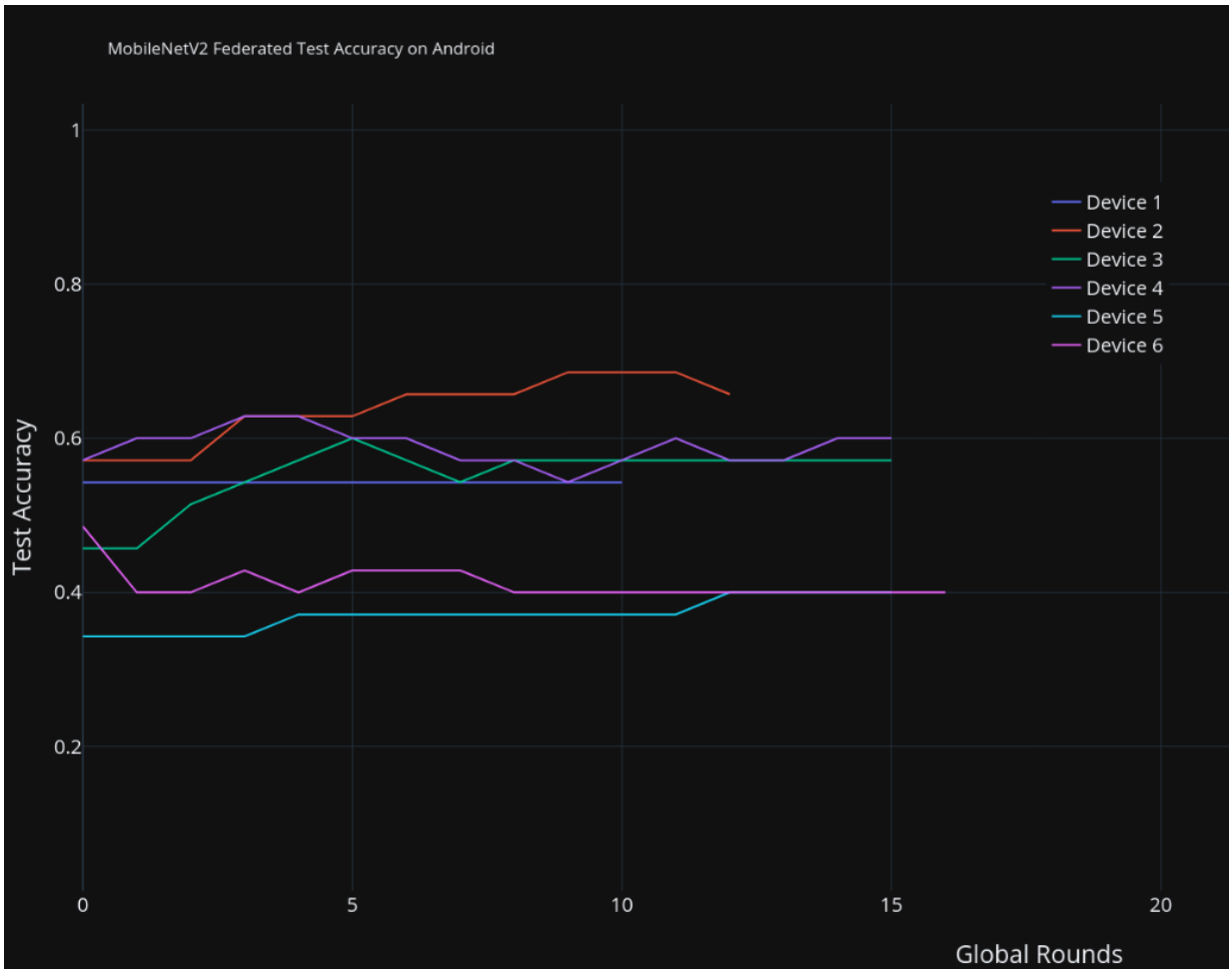


Figure 17. MobileNetV2 Test Accuracy plot for Android devices.

Figure 17 shows the initial average Test Accuracy is 49.52% from 6 devices, and after 20 global rounds of training with 20 local epochs for each device, the final average Test Accuracy reached 52.86%. This increase in Test Accuracy is greater than that of VGGFace. Besides, Figure 17 shows three of the devices had a consistently improving learning curve.

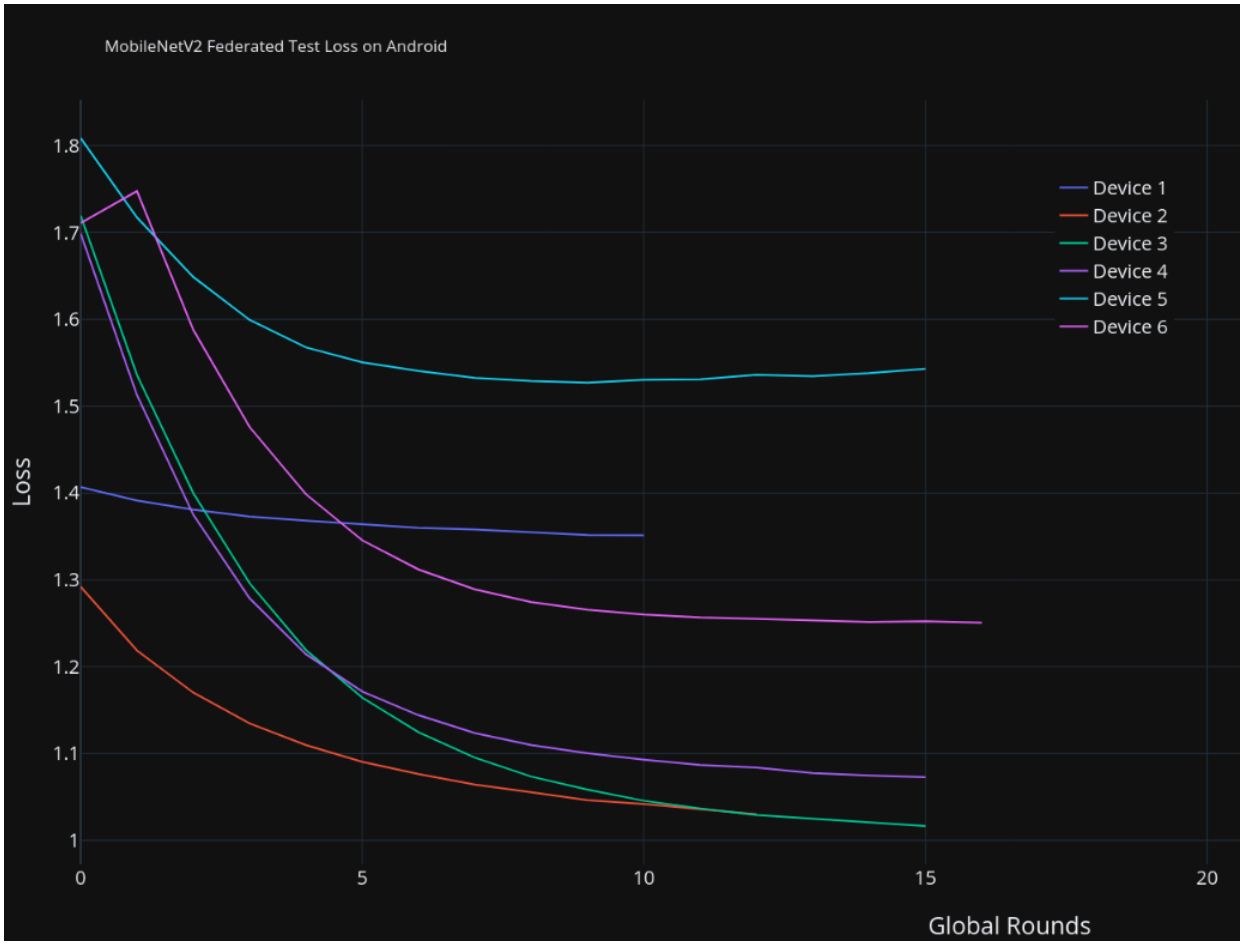


Figure 18. MobileNetV2 Loss plot for Android devices.

Figure 18 demonstrates that the average loss value decreased from 1.61 to 1.21 across 6 Android devices throughout the training process. A smaller loss value means the model is making fewer prediction errors and it proves the MobileNetV2 is learning correctly with improvements.

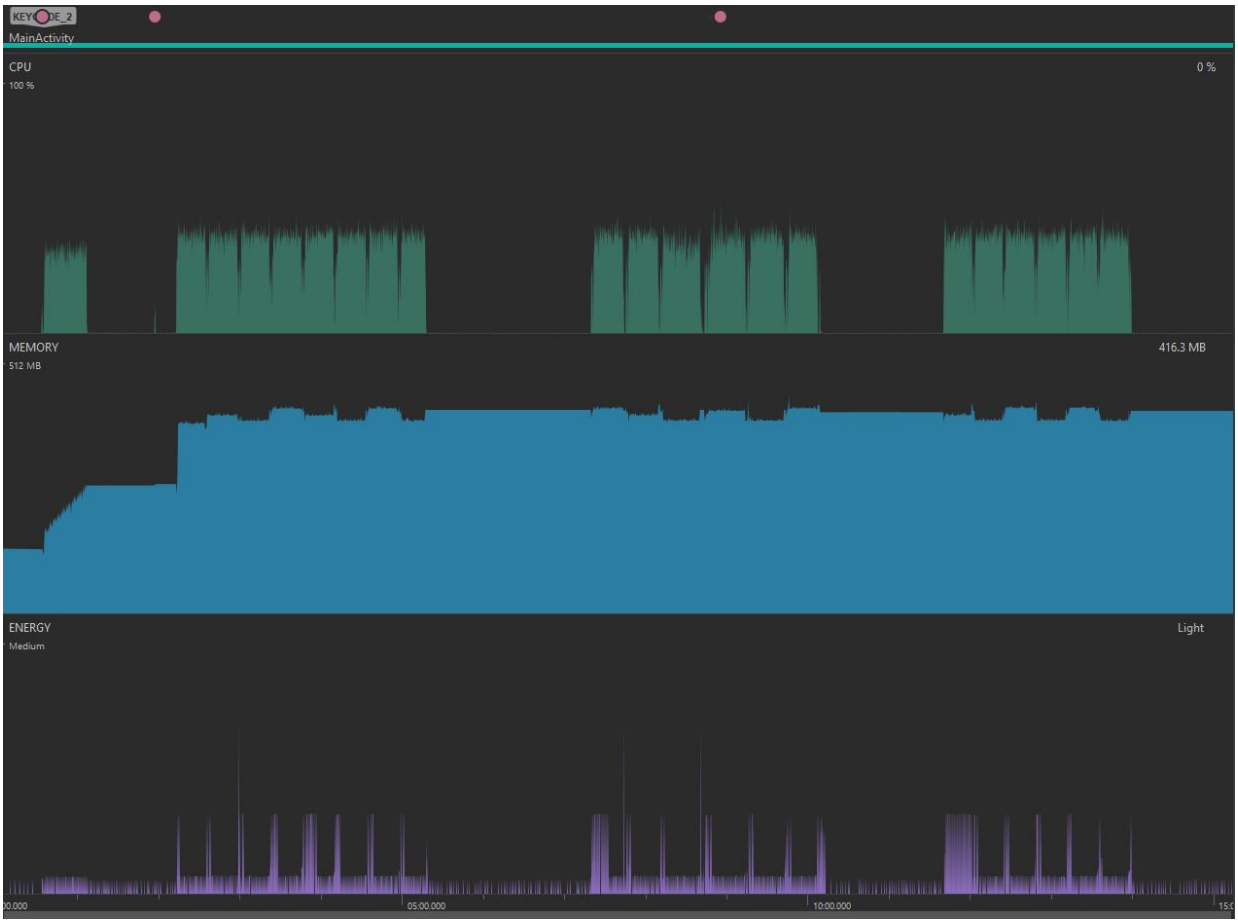


Figure 19. Android resource usage monitoring during MobileNetV2 training.

Figure 19 shows that the CPU usage was well below 45% during the image loading and training phases. Memory Usage was about 416 MB and energy consumption stayed low. In comparison to VGGFace, MobileNetV2 requires less than half of the CPU and energy usage.

4.3.4 *Test Result Comparison*

To evaluate the federated model performance, we compared the accuracy, loss, size, training time, and Android resource usage. We will only compare VGGFace and MobileNetV2 in this section, as ResNet50 is too heavy to run and produce any test data.

Table 12. Model test results in Flower Federated.

	VGGFace Lite	ResNet50 Lite	**MobileNetV2 Lite**
Initial Accuracy	50.48%	N/A	**49.52%**
Final Accuracy	51.90%	N/A	**52.86%**
Initial Loss	1.28	N/A	**1.61**
Final Loss	Infinity	N/A	**1.21**
Model Size	64.1 MB	103 MB	**19.4 MB**
Training Time	1067.5 seconds	N/A	**723.2 seconds**
CPU Usage	65% - 100%	N/A	**< 45%**
Memory Usage	430 MB	N/A	**416 MB**
Energy Consumption	Heavy	N/A	**Low**

Table 12 shows that MobileNetV2 learned new knowledge properly, with a larger accuracy improvement and a better loss reduction. Due to its compact size, it also used less time and lighter resource requirements to complete training. In summary, MobileNetV2 learned better, faster, and more efficiently in the federated learning setting compared to VGGFace.

4.4 STAGE-BY-STAGE EVALUATION

In this section, we will evaluate the performance of VGGFace Version 1, VGGFace Version 2, ResNet50, and MobileNetV2 in the three stages based on the Evaluation Metrics defined in Section 3.6. In Stage 1, we can see that between these model architectures, the more parameters a model has, the larger the peak accuracy is and the heavier the model is. In addition, using pre-trained facial weights significantly sped up the training time with faster convergence. However, it did not further increase peak accuracy compared to a model with randomly initialized weights. The highest model accuracy we achieved in Stage 1 is 62.07% using ResNet50, which is a reasonable result for a 7-race classification problem. As a reminder, Kärkkäinen K. and Joo J. [27] used 4 races

(White, Black, Asian, and Indian) from FairFace and achieved 75.4% accuracy to identify non-White faces with the ResNet34. With more racial groups included in our project, there is more complex and refined feature information for our models to differentiate and recognize.

In Stage 2, we can see that the TF Lite Converter can significantly reduce the size of a heavy CNN model by more than 60%. However, such models are usually still much larger than a lightweight model like MobileNetV2. Lighter models are likely a more viable candidate, in general, to deploy on edge devices where on-device training can be enabled.

In Stage 3, it is worth noting that heavy models like VGGFace and ResNet50 are not feasible for federated training on Android even after the TF Lite conversion. These heavy models can lead to improper training or cause the application to crash, as Android devices have limited memory capacity. On the other hand, MobileNetV2 proved to be a reliable solution for fast and lightweight on-device training for Flower. Not only was accuracy improved, but the training required less time and resources to complete. In this project, MobileNetV2 was able to use the knowledge learned in Stage 1, and further expand its knowledge base through correct and efficient federated training in Stage 3.

Chapter 5. CONCLUSION

In this project, we proposed a novel Facial Recognition FL system using CNN models and Flower Federated. we examined four major components: dataset, Stage 1 model preparation, Stage 2 model reduction, and Stage 3 model testing. We implemented four model architectures and experimented with different hyperparameter settings in a trial-and-error manner to achieve the best performance possible. While VGGFace and ResNet50 have higher accuracy for face classification in Stage 1 centralized setting, we considered the performance-weight tradeoff and proved that theory in Stage 3 federated testing. MobileNetV2 with the lowest accuracy from Stage 1 was the only model that successfully completed 20 global rounds of federated training in Stage 3. On the other hand, VGGFace and ResNet50 failed to enable on-device training with Flower Android. Our results show that in Stage 3, MobileNetV2 can continue learning new information efficiently and effectively. Although the final accuracy is 52.86% and lower than a similar study done on 4 races, we consider our results to be acceptable as our classification includes 7 races and a lightweight model must be selected for Android deployment. In the meantime, our system has the novelty of using actual Android devices for FL implementation and testing, as most previous studies used powerful computers to simulate FL clients and avoid hardware restriction overhead. Our FL system proves to be a viable solution to preserving user privacy because private face data is separate from the central server in the entire training process. The model setup and pipeline should be able to also apply to other image classification tasks to solve similar sensitive problems.

Chapter 6. FUTURE WORK

As the server in a FL system does not have knowledge of user inputs, in a real-world FL scenario this nature poses vulnerability to malicious user input or labeling. There should be algorithms in the Android application to prevent users from uploading images that are low-quality, blurry, poorly lit or framed. Furthermore, we can deploy a lightweight model to first detect if the input image contains a human before the device is allowed to participate in the training. Additional filtering can be implemented to return a confidence score in whether the image is correctly labeled to its race. These steps can add security against adversarial attacks as well. Lastly, the application GUI can be further improved to show more information about the user's involvement and contribution to the training process.

BIBLIOGRAPHY

[1] G. Gottsegen, "16 Machine Learning Examples Your Industry Needs to Know Now," builtin, 10 December 2021. [Online]. Available: https://builtin.com/artificial-intelligence/machine-learning-examples-applications.

[2] D. Shewan, "10 Companies Using Machine Learning in Cool Ways," WordStream, 3 December 2021. [Online]. Available: https://www.wordstream.com/blog/ws/2017/07/28/machine-learning-applications.

[3] C. D. W., "Digital images are data: and should be treated as such.," *Methods in molecular biology,* vol. 931, pp. 1-27, 2013.

[4] J. Brownlee, "9 Applications of Deep Learning for Computer Vision," Machine Learning Mastery, 13 March 2013. [Online]. Available: https://machinelearningmastery.com/applications-of-deep-learning-for-computer-vision/.

[5] A. Najibi, "Racial Discrimination in Face Recognition Technology," Harvard University The Graduate School of Arts and Sciences, 24 October 2020. [Online]. Available: https://sitn.hms.harvard.edu/flash/2020/racial-discrimination-in-face-recognition-technology/.

[6] E. S. AB, "ekkono.ai," May 2020. [Online]. Available: https://www.ekkono.ai/wp-content/uploads/2020/12/SWP_Federated_Learning_Ekkono_Solutions_May_2020.pdf.

[7] B. McMahan and D. Ramage, "Google AI Blog," Alphabet, 6 April 2017. [Online]. Available: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[8] P. R. a. G. B. Nicol Turner Lee, "Algorithmic bias detection and mitigation: Best practices and policies to reduce consumer harms," 22 May 2019. [Online]. Available: https://www.brookings.edu/research/algorithmic-bias-detection-and-mitigation-best-practices-and-policies-to-reduce-consumer-harms/.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke,

Y. Yu and X. Zheng, "TensorFlow," Google, 2015. [Online]. Available: https://www.tensorflow.org/.

[10] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet and N. D. Lane, "Flower: A Friendly Federated Learning Research Framework," *CoRR*, vol. abs/2007.14390, 2020.

[11] S. Albawi, T. A. Mohammed and S. Al-Zawi, "Understanding of a convolutional neural network," *IEEE*, pp. 1-6, 2017.

[12] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way," towardsdatascience, 15 December 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[13] M. A. Abu, N. H. Indra, A. Abd Rahman, N. Sapiee and I. Ahmad, "A study on Image Classification based on Deep Learning and Tensorflow," vol. 12, p. 04, 2019.

[14] A. Gandhi, "Data Augmentation | How to use Deep Learning when you have Limited Data—Part 2," Nanonets, 2021. [Online]. Available: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/.

[15] M. Hussain, J. Bird and D. Faria, "A Study on CNN Transfer Learning for Image Classification," 2018.

[16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *CoRR*, vol. abs/1512.00567, 2015.

[17] F. Sultana, A. Sufian and P. Dutta, "Advancements in Image Classification using Convolutional Neural Network," *CoRR*, vol. abs/1905.03288, 2019.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248-255, 2009.

[19] C. a. B. S. a. H. M. a. R. B. a. V. O. Zhang, "Understanding deep learning requires rethinking generalization," *arXiv*, 2016.

[20] J. Nabi, "Hyper-parameter Tuning Techniques in Deep Learning," towardsdatascience, 16 March 2019. [Online]. Available: https://towardsdatascience.com/hyper-parameter-tuning-techniques-in-deep-learning-4dad592c63c8.

[21] D. E. King, "Dlib-ml: A Machine Learning Toolkit," *Journal of Machine Learning Research,* vol. 10, pp. 1755-1758, 2009.

[22] K. Zhang, Z. Zhang, Z. Li and Y. Qiao, "Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks," *CoRR,* vol. abs/1604.02878, 2016.

[23] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools,* 2000.

[24] J.-C. Chen, V. M. Patel and R. Chellappa, "Unconstrained Face Verification using Deep CNN Features," *CoRR,* vol. abs/1508.01722, 2015.

[25] Q. Cao, L. Shen, W. Xie, O. M. Parkhi and A. Zisserman, "VGGFace2: A dataset for recognising faces across pose and age," *CoRR,* vol. abs/1710.08092, 2017.

[26] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," *CoRR,* vol. abs/1512.03385, 2015.

[27] K. Kärkkäinen and J. Joo, "FairFace: Face Attribute Dataset for Balanced Race, Gender, and Age," *CoRR,* vol. abs/1908.04913, 2019.

[28] J. G. a. J. Sarlin, "A false facial recognition match sent this innocent Black man to jail," CNN Business, 29 April 2021. [Online]. Available: https://www.cnn.com/2021/04/29/tech/nijeer-parks-facial-recognition-police-arrest/index.html.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024-8035.

[30] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *CoRR,* vol. abs/1512.01274, 2015.

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

[32] A. Mashhadi, B. Lagesse and M. Stiber, "FAI: Crowd-sourcing Fairness for Demographic Detection in Convolutional Neural Networks," University of Washington, Bothell, 2022.

[33] Chollet, Francois and others, "Keras," GitHub, 2015. [Online]. Available: https://github.com/fchollet/keras.

[34] O. M. a. V. A. a. Z. A. Parkhi, "Deep Face Recognition," in *BMVC*, 2015.

[35] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation," *CoRR*, vol. abs/1801.04381, 2018.

[36] S. Park, "A 2021 Guide to improving CNNs-Optimizers: Adam vs SGD," Medium, 20 June 2021. [Online]. Available: https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008.

[37] X. Wang, H. Zhao and J. Zhu, "GRPC: A Communication Cooperation Mechanism in Distributed Systems," *SIGOPS Oper. Syst. Rev.*, vol. 27, pp. 75-86, 1993.

[38] H. B. McMahan, E. Moore, D. Ramage, S. Hampson and B. A. y. Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," *CoRR*, vol. abs/1602.05629, 2016.

[39] J. Guo, Z. Liu, K.-Y. Lam, J. Zhao, Y. Chen and C. Xing, "Secure Weighted Aggregation for Federated Learning," *CoRR*, vol. abs/2010.08730, 2020.

[40] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," *CoRR*, vol. abs/1511.08458, 2015.