POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING

Master's Degree Thesis

Recognizing human activities in a privacy-preserving way

Supervisors

Prof. Alessio SACCO

Prof. Guido MARCHETTO

Candidate

Bruno PALERMO

October 2024

Acknowledgements

This is an achievement that I owe to all the people who have been close to me during these intense years, both here and up above. Family, friends, and figures who have played a fundamental role in my growth, both personal and academic. Special thanks to mamma Maria, papà Nunzio and my siblings, Enrico and Giulia, who have always provided me with a safe haven, a place I am lucky enough to call Ohana.

To my dear friends: Chiara, Giulia and Sebastiano, to whom I owe countless stories over the years. You have been the light that has illuminated my experience in Turin. To Marco, who like a brother has shared these five years with me, a true right-hand man.

Thanks to Doriana, my roommate on the spectral Corso Racconigi house and fellow compaesana, and to the guys on the fourth floor of the Borsellino with whom I spent unforgettable Sunday lunches.

To Filippo, a Piedmontese colleague who made me discover the mountains as a Sicilian. From him, I learnt that long slopes only precede breathtaking peaks. To Eliza, an extraordinary person, for the sleepless nights we spent studying together and for her constant support. A thought also goes to the guys from Trapani, sincere friends, that despite the distance, have always encouraged me along my path and to all the summer days we shared.

Table of Contents

List of Tables VII

List of Figures VIII

Acronyms XII

1 Introduction 1

1.1 Motivation . . . 1
1.2 Objective . . . 2
1.3 Thesis structure . . . 2

2 Background 3

2.1 Human Activity Recognition . . . 3
2.1.1 HAR problem . . . 4
2.1.2 Problem definition . . . 4
2.1.3 Relaxed Problem Definition . . . 4
2.1.4 HAR process . . . 5
2.1.5 Data acquisition . . . 5
2.1.6 Data preprocessing . . . 6
2.1.7 Model training . . . 9
2.1.8 Activity Classification . . . 13
2.2 Introduction to Federated Learning . . . 14
2.2.1 Flower Framework . . . 14
2.2.2 gRPC . . . 15
2.3 Introduction to Transfer Learning . . . 16
2.3.1 Tensorflow . . . 16
2.3.2 TF Lite . . . 17
2.3.3 Post-Training Quantization . . . 17

3 Related Work 19

3.0.1 HAR in smartphone devices: . . . 19

3.0.2 Federated Learning: 20
3.0.3 Transfer Learning 21
3.0.4 Federated Transfer Learning (FTL) 22

4 System 23
4.1 System Overview 23
4.2 PHAR Android Application 23
4.2.1 Project overview 26
4.2.2 Real-Time Sensor Monitoring 26
4.2.3 Test Configuration 28
4.2.4 Local Testing 28
4.2.5 Federated Testing 31
4.2.6 TensorFlow Library 32
4.3 Flower Learning Server Setup 35
4.3.1 How Flower Server and Clients Collaborate 35

5 Experimental results 38
5.1 Data acquisition 38
5.2 Data preprocessing 40
5.2.1 Resampling 40
5.2.2 Normalization 40
5.2.3 Data Segmentation 42
5.3 Model 45
5.3.1 Post-training quantization 48
5.4 Classifications and Evaluations 49

6 Conclusions 59

Appendix 61

Bibliography 69

List of Tables

5.1 Overview of the datasets. A: Accelerometer, G: Gyroscope, M: Magnetometer . . . 39
5.2 Class distribution of samples in the MS dataset. . . . 39
5.3 Class distribution of samples in the MA dataset. . . . 40
5.4 Summary statistics of various attributes in the dataset before normalization. . . . 41
5.5 Summary statistics of various attributes in the dataset MA before normalization. . . . 42
5.6 Class-wise sample counts and percentages per partition, including overall partition totals and their percentage of the entire MS. P: partition . . . 43
5.7 Class-wise sample counts and percentages per partition, including overall partition totals and their percentage of the entire MA. P: partition . . . 44
5.8 Baseline performance metrics for the MA-MA and MS-MS datasets. 45
5.9 Resulting metrics for the combination of datasets . . . 46
5.10 Baseline performance metrics for the MA-MA and MS-MS datasets. 50
5.11 Resulting metrics for the combination of datasets . . . 51
5.12 Performance metrics for various dataset combinations (Samples: 600) 52
5.13 Comparison of Training Time (TT) between No-FL and FL for Python version. . . . 56
5.14 Comparison of Accuracy and F1-Score between No-FL and FL for Python version. . . . 57

List of Figures

2.1 Aliasing . . . 7
2.2 Data segmentation . . . 8
2.3 Convolution layer computation . . . 11
2.4 Max-pooling layer computation . . . 12
2.5 Softmax example . . . 13

4.1 System overview . . . 24
4.2 Overview . . . 27
4.3 Real-Time sensor monitoring section . . . 28
4.4 Test configuration overview . . . 29
4.5 Transfer Learning Overview . . . 29
4.6 Transfer Learning notification . . . 31
4.7 Federated Learning Notification . . . 32
4.8 Federated Learning Overview . . . 33
4.9 Overview of the Flower Server and client Interaction from [48] . . . 36

5.1 Activity recognition process . . . 38
5.2 Frequency response of the Butterworth low-pass filter. . . . 41
5.3 Impact of Range normalization on model accuracy . . . 43
5.4 Impact of MinMax normalization on model accuracy . . . 43
5.5 Impact of Robust normalization on model accuracy . . . 43
5.6 Impact of Z-Score standardization on model accuracy . . . 43
5.7 Comparison of the effects of different normalization techniques on model accuracy over multiple samples settings. . . . 43
5.8 Effect of Robust normalization on model accuracy with Adam optimizer . . . 44
5.9 Effect of Robust normalization on model accuracy with RMSProp optimizer . . . 44
5.10 Effect of Z-Score standardization on model accuracy with Adam optimizer . . . 44

5.11 Effect of Z-Score standardization on model accuracy with RMSProp optimizer . . . 44
5.12 Comparison of the effects of different normalization techniques on model accuracy across various sample settings and hyperparameters. 44
5.13 Comparison of model performance across different sample configurations and clients (Android vs. Python). . . . 46
5.14 Effect of quantization on the model size . . . 48
5.15 TL: Classification metrics vs. Number of Pre-trained Models for Android version . . . 53
5.16 TL: Classification metrics vs. number of pre-trained models for python version . . . 53
5.17 TL: Training time vs. number of pre-trained models for Android version . . . 54
5.18 TL: Training time vs. number of pre-trained models for Python version . . . 54
5.19 TL: Accuracy for quantization method averaged on dataset combinations using pre-trained layers . . . 55
5.20 TL: F1-Score for quantization method averaged on dataset combinations using pre-trained layers . . . 55
5.21 TL: Training time for quantization method averaged on dataset combinations using pre-trained layers for Android version. . . . 56
5.22 FTL: Comparison of accuracy for Android and Python versions . . . 57
5.23 FTL: Comparison of f1-score for Android and Python versions . . . 58

1 Performance of 600 samples configuration with no pretrained layers before FL-TL . . . 61
2 Performance of 2000 samples configuration with no pretrained layers before FL-TL . . . 61
3 Performance of 2500 samples configuration with no pretrained layers before FL-TL . . . 62
4 Performance of 3000 samples configuration with no pretrained layers 62
5 Performance of 4500 samples configuration with no pretrained layers before FL-TL . . . 63
6 Non normalized attributes distribution of MotionSense dataset . . . 63
7 Normalized attributes distribution of MotionSense dataset . . . 64
8 Non normalized attributes distribution of MobiAct dataset . . . 64
9 Normalized attributes distribution of MobiAct dataset . . . 65
10 The initialization file setting up the initial parameters. . . . 65
11 The bottleneck model. . . . 66
12 The optimizer file utilized for optimization . . . 66
13 The inference file generating predictions. . . . 67

14 The training head model 67
15 A customized version of the model proposed in [51]. 68

Acronyms

Adam
: Adaptive Moment Estimation

AI
: Artificial Intelligence

CNN
: Convolutional Neural Network

DL
: Deep Learning

FC
: Fully Connected

FedAvg
: Federated Averaging

FL
: Federated Learning

GDPR
: General Data Protection Regulation

gRPC
: Google Remote Procedure Call

HAR
: Human Activity Recognition

MCU
: Microcontroller Unit

MEMS
: Micro-Electro-Mechanical Systems

ML
: Machine Learning

PTQ
: Post-Training Quantization

ReLU
: Rectified Linear Unit

RPC
: Remote Procedure Call

SGD
: Stochastic Gradient Descent

TF
: Tensorflow

TFLite
: TensorFlow Lite

TL
: Transfer Learning

Chapter 1

Introduction

1.1 Motivation

The rapid evolution of ML is transforming industries and reshaping entire research fields. Among these advancements, HAR stands out as a promising area that bridges human activities and ML technologies. HAR involves the identification of physical activities such as jumping, running, or sleeping using data captured from sensors that might be embedded in ubiquitous devices like smartphones[1]. and watches. This capability has significant applications in fields ranging from healthcare [2, 3], and surveillance [4] to smart environments [5].

In today's privacy-conscious world, the challenge is not only to advance in ML to support data-driven decision-making, predictive analysis, environmental impact assessments, and cost savings, but also to do so in a way that safeguards user privacy and complies with regulations like the General Data Protection Regulation (GDPR) and adheres to data minimization principle [6] for the privacy of consumer data. Preserving privacy is essential to mitigate security risks when handling personal and sensitive data. It is especially critical in applications like HAR, where ensuring accurate predictions and minimize centralizing user data is necessary due to the sensitivity of the information involved.

Moreover, computational and storage limitations, particularly in mobile and edge devices, present an additional layer of complexity[7]. With these constraints in mind, there is an increasing demand for efficient ML techniques that minimize the amount of data required for training while still providing rapid and reliable predictions.

Transfer Learning proves to be an efficient technique in this scenario. Rather than building models from scratch, it enables the reuse of pre-trained models, which can be used to related tasks, even when using different data sources. This method greatly reduces the amount of training required and accelerates deployment,

making it particularly well-suited for resource-constrained environments like mobile applications in HAR by minimizing the need for extensive training data.

Federated Learning enables decentralized model training across multiple devices without transferring personal data to a central server[8]. In a FL setting, each device—such as a smartphone—trains the model locally on its own data, and only the updated model weights are shared with the central server. This ensures data privacy while still improving the global model's performance. FL is particularly relevant for HAR, where users' activity data is inherently personal and sensitive, and traditional centralized approaches would raise privacy concerns as in the case of smart environments. [5].

1.2 Objective

This thesis explores how various techniques can converge on a device already in the hands of billions: the smartphone. By utilizing the sensors built into smartphones, it is possible to gather the data needed for HAR while safeguarding user privacy and addressing the computational constraints of mobile environments.

Overcoming these challenges requires a multi-dimensional approach. First, creating an environment that builds on prior challenges by following the steps outlined in the background section. Next, investigating how to deploy models on smartphones using existing tools, while understanding and addressing their limitations. Finally, integrating privacy-preserving methods like FL ensures that progress in accuracy and efficiency does not come at the cost of user security or data privacy.

1.3 Thesis structure

The thesis is structured to offer a comprehensive understanding of its objectives, beginning with an explanation of the fundamental concepts and progressing through the implementation, the experimental tests and final results. It starts with an analysis of the foundational background, outlining theories and techniques. This is followed by a detailed review of the related work being conducted on the field, highlighting critical observations and how this thesis seeks to address challenges and drive further progress. The next section provides an overview of the experimental system, including the setup, results, and analysis. Finally, the thesis concludes with a reflection on the insights gained, observations, and recommendations for future research directions.

Chapter 2

Background

Before introducing the solution that has been developed and tested, it is important to clarify some of the concepts that will be mentioned. This will provide the necessary context to comprehensively understand what will be discussed in the next chapters. Among the concepts, the following three key areas of focus in this thesis are outlined:

- **Human Activity Recognition**
- **Federated Learning**
- **Transfer Learning**

2.1 Human Activity Recognition

A noteworthy research field due its applications in numerous interdisciplinary study area is HAR. By analyzing a sequence of observations on human actions within certain environmental conditions, HAR aims to identify the activity being performed.

Given its multifaceted nature, HAR can be approached through various strategies, including logic and reasoning, probabilistic models, data mining or ML. ML, in particular, involves discovering patterns and relationships using algorithms and models. This approach is particularly useful for identifying subtle or complex patterns in data, such as sensor readings associated with activities like walking, standing, running, and sitting, which might not be immediately apparent.

Data for HAR can be collected from various sources, including social media, video surveillance [4], or raw sensor data. This section will define the HAR problem specifically within the context of raw sensor data, outlining the core concepts involved.

2.1.1 HAR problem

2.1.2 Problem definition

Activities are complex events that can be structured using a formal approach. Consider a set $S = \{S_0, \ldots, S_{k-1}\}$ of k time series, where each series represents measurements of a particular attribute over a time interval I. The goal is to partition this interval into consecutive segments $I_0, \ldots, I_{r-1}$, each labeled with the activity being performed. These intervals must satisfy several conditions: they must be consecutive, non-empty, each represent only one activity, and their union must cover the entire time span I [9].

Solving this problem deterministically is infeasible due to the vast (potentially infinite) number of attributes that can influence activity recognition. Moreover, in real-world scenarios, assigning a single activity to each time interval is often unrealistic. For instance, running can occur while holding a phone or breathing, resulting in multiple overlapping attributes that describe the combination of these activities. Accurately identifying the activity of "running" may require considering both acceleration and location data, to give an example. However, in cases such as running on a treadmill, location data may not provide sufficient information, since it remains constant, while acceleration alone might struggle to distinguish running from other activities, such as swimming. To cope with these challenges, constraints must be relaxed and approximations introduced to account for overlapping and ambiguous signals.

2.1.3 Relaxed Problem Definition

To address the limitations of the previous formulation, a relaxed version of the problem is defined. The goal is to find a mapping function $f : S_i \rightarrow A$, such that $f(S_i)$ closely approximates the actual activity performed during each interval I_i. In this relaxed approach, it is acknowledged that during transition periods between activities, multiple activities can overlap within a single time window. To capture these overlapping patterns at the boundaries of time intervals, overlapping windows can be introduced. This approach allows the model to better handle ambiguous signals that occur at activity transitions.

Furthermore, the set of activities A is defined as a set $A = \{a_0, \ldots, a_{n-1}\}$ of activity labels, where each label corresponds to a distinct activity. By using this relaxed approach, the model can more effectively deal with the inherent complexity of human activities, reducing the error introduced by sharp transitions between activities. The overlapping windows help mitigate the ambiguity that arises when multiple activities occur within a single time window, providing a smoother and more accurate representation of the activity being performed.

2.1.4 HAR process

There are 4 steps that can be found in the HAR process:

- **Data acquisition**: this involves collecting raw data from sensors, such as accelerometer, gyroscope and magnetometer. The data is the representation of the physical movement made by the subject of the study which is associated with the human activity.

- **Data preprocessing**: raw sensor data is cleaned and transformed to prepare it for analysis. This step includes noise reduction, data filtering, normalization, segmentation into meaningful time intervals, and feature extraction.

- **Model training**: this step involves teaching a ML model to recognize and classify activities based on preprocessed data. The process includes choosing an appropriate algorithm (e.g., decision trees, support vector machines, or neural networks) and tuning its hyperparameters to optimize performance.

- **Activity classification**: after the model is trained, it is used to classify activities based on new data. The model processes the features extracted from incoming data and generates predictions of the activity being performed.

2.1.5 Data acquisition

Given the outline of the HAR process, it is important to introduce smartphone sensors to highlight their advantages. These benefits will be explored in detail in the subsequent chapters.

Smartphone sensors

Smartphones have become indispensable in our daily lives. Originally invented in 1849 by the Italian innovator Antonio Meucci, the telephone has undergone remarkable evolution. Within a century, phones began incorporating sensors, transforming them into the smart, interactive devices we rely on today.

These powerful innovations have become so pervasive that by 2020, over 5 billion phones will be in use worldwide [10]. Each device is equipped with advanced sensors like accelerometers, gyroscopes, and GPS, which continuously monitor various user activities, including physical movements. Their widespread usability is driven by their affordability and availability in the market.

The sensors embedded in smartphones, such as gyroscopes and accelerometers, are classified as transducers because they convert energy from one form to another. Typically made using Micro-Electro-Mechanical Systems (MEMS) technology, these miniaturized mechanical and electro-mechanical elements perform essential

mechanical functions, enabling parts of the sensor to move and interact with the environment. The mechanical energy detected is converted into electrical signals, which is then processed at regular intervals. This processing involves amplification, digitization, and transmission to the system for visualization.[11]

2.1.6 Data preprocessing

Preprocessing is a fundamental step to improve the quality of the sensor data. A general preview of the subsequent steps is illustrated.

Resampling

To ensure comparability across data sources, it is essential that all data be sampled at the same frequency. When resampling, whether through interpolation or aggregation, a low-pass filter must be applied first. Without this, decimating (reducing the sampling rate) a signal can introduce undesirable aliasing. Aliasing occurs when higher frequency components are misrepresented as lower frequencies, leading to distortion. To prevent this, decimation must always be combined with an anti-aliasing filter. For clarity, the example 2.1 illustrates this process in more detail.

Normalization

Sensor readings can be influenced by various factors, including sensor characteristics and environmental conditions. To ensure consistent and comparable data, it's essential to first remove outliers that could skew the normalization process. After outliers are addressed, scaling features to a common range normalizes the data, making it more uniform across different sensors and conditions. Common normalization techniques include Min-Max Scaling, and Robust Scaling or standardization techniques like Z-Score Standardization. The choice of normalization technique depends on factors such as the presence of outliers, the distribution of the dataset, and the domain.[12]

The following is a list of the most commonly used normalization functions employed in this study.

Z-Score Standardization

$$Z = \frac{x - \mu}{\sigma} \tag{2.1}$$

where:

- x is the data point,
- μ is the mean of the dataset,

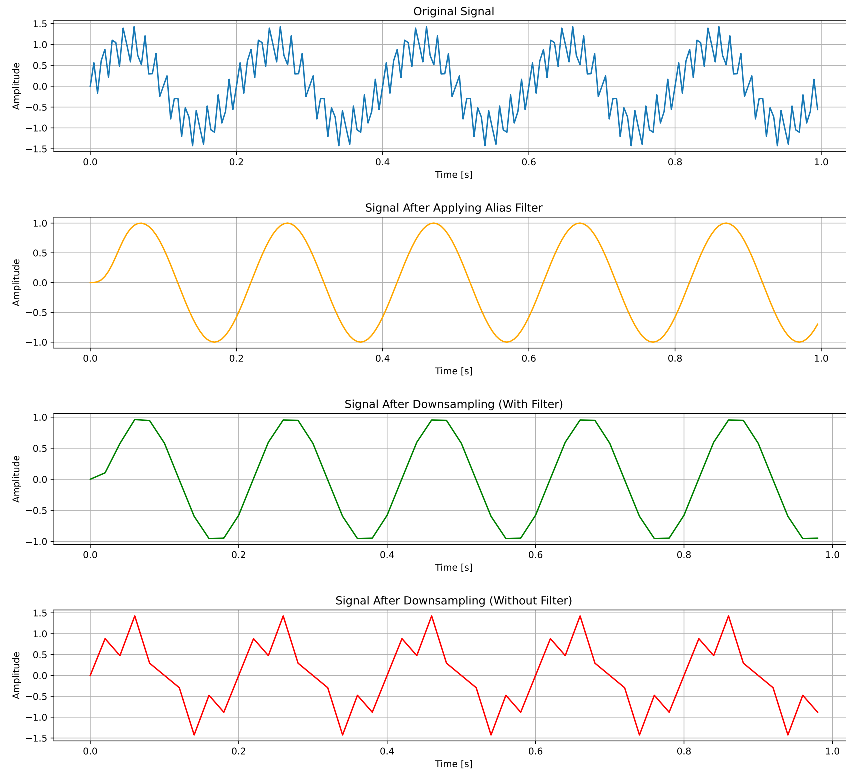

Figure 2.1: Aliasing

- σ is the standard deviation of the dataset.

Min-Max Normalization

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \tag{2.2}$$

where:

- x is the original data point,
- x_{min} is the minimum value of the dataset,
- x_{max} is the maximum value of the dataset.

Range Normalization

$$x' = a + \frac{(x - x_{\text{min}})(b - a)}{x_{\text{max}} - x_{\text{min}}} \tag{2.3}$$

where:

- a and b define the desired range,
- x is the original data point,
- x_{min} is the minimum value of the dataset,
- x_{max} is the maximum value of the dataset.

Robust Normalization

$$x' = \frac{x - Q_1}{Q_3 - Q_1} \tag{2.4}$$

where:

- Q_1 is the first quartile (25th percentile),
- Q_3 is the third quartile (75th percentile),
- x is the original data point.

Data Segmentation

In the next step, the data are processed into sections known as segments. The size of each segment is crucial, as it influences the model's performance in subsequent stages. Larger segments may require more computational resources and might not effectively capture specific activities, while smaller segments may be too granular, potentially failing to represent the activity accurately. [13, 14, 15].

A common approach for data segmentation is the sliding window technique, where time-series data is divided into overlapping segments based on a defined window size. In this method, the time-series data are split into segments with overlapping regions determined by a window step size. The choice of window size and step size depends on factors such as data availability, the need to capture patterns at boundaries, computational cost, and redundancy.

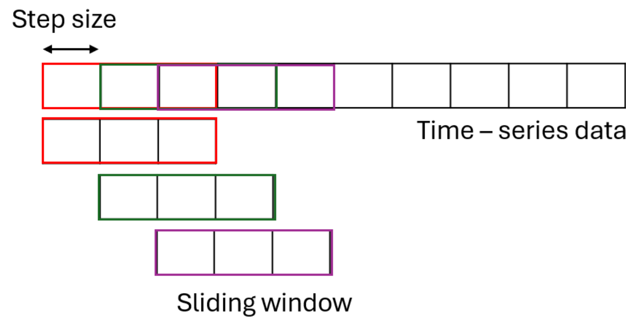

Figure 2.2: Data segmentation

Figure 2.2 illustrates an example where a single time-series array representing the study of a specific attribute is segmented using a sliding window approach. Here, a defined window size captures a particular time frame, and this window is then applied across the data, allowing for overlapping segments.

2.1.7 Model training

In the next step, it is necessary to identify a model capable of handling sensor data effectively. A model encapsulates the core logic required for making predictions based on the processed data. It must be contextualized within the specific application and requirements of the HAR system. In the following, a brief introduction to deep learning, focusing on its relevance and application in the context of HAR. This overview will lay the groundwork for the detailed discussions in the subsequent and the chooses made.

Introduction to Deep Learning

ML is a broad field focused on developing systems that can learn and adapt without explicit programming for each specific task. Central to ML are algorithms and models, which define the methods and approaches used to enable this learning.

One area within ML is *Deep Learning*, which utilizes complex models known as neural networks. These models are inspired by the structure and functioning of the human brain, specifically its interconnected neurons.

Neuron

A neuron is a specialized cell in the brain that communicates through a combination of chemical and electrical signals. They transmit information across synapses—gaps between neurons—using electrical impulses called action potentials. Neurons receive multiple inputs, process this information, and then transmit signals to other neurons in an evolving pattern. [16]

Neural Networks

Neural networks consist of layers of interconnected nodes or neurons that process data in a manner similar to the brain's own processing system. With limited human assistance, they can learn and model the relationship between input and output data that are nonlinear and complex.

Each node in the network is linked to others with a specific weight and threshold. If a node's output surpasses its threshold, the node activates and sends information to the next layer. If the output falls below the threshold, the node stays inactive, and no information is passed to the next layer.

A basic neural network has interconnected neurons in three layers:

- **Input Layer**: Receives various features from the data source, process and categorize passing to next layers.
- **Hidden Layer**: Intermediate layer where data is processed through weighted connections and activation functions. This layer perform complex transformations on the data.
- **Output Layer**: Produces the final predictions or classifications based on the processed data. For instance, in a classification task with multiple categories, the output layer might use a dense layer with several nodes (one for each category for multi-class classification problems), often employing techniques like one-hot encoding to represent different classes.

Neural networks have been proven to be effective in various industries where they find application. In the medical field, they enhance image classification for diagnostic purposes. In social networks, they refine target marketing strategies. In finance, they aid in making precise financial predictions. In the energy sector, they improve electrical load forecasting.

Convolutional Neural Networks

Neural networks can be classified based on how data flows from the input node to the output node. One category is the feedforward neural network, which is the simplest type of neural network. In a feedforward network, neurons are arranged in layers that process data sequentially, with no feedback loops.

A CNN is a specialized form of feedforward neural network designed to automatically learn features through the optimization of filters (or kernels). By employing regularized weights and limiting connections, CNNs effectively address issues like vanishing and exploding gradients.

In CNNs, hidden layers carry out specific mathematical operations, such as summarization or filtering, known as convolutions. These operations are particularly effective for tasks like classification and computer vision. CNNs utilize linear algebra techniques, such as matrix multiplication, to detect patterns within for example images. [17]

The primary layers in a CNN include the convolutional layer, pooling layer, and fully connected (FC) layer. The convolutional layer is typically the first, and it may be followed by additional convolutional or pooling layers. The final layer is a fully connected layer.

Convolutional Layer

The convolutional layer is the building block of a CNN. It operates by applying a filter (or kernel) to the input data. The filter, a set of weights, slides across the input data, performing a dot product at each position to produce an output known as the *feature map* or *activation map*. This process is called convolution. Visually, it can be seen as in the figure 2.3 Consider an input matrix of size 6×6. The

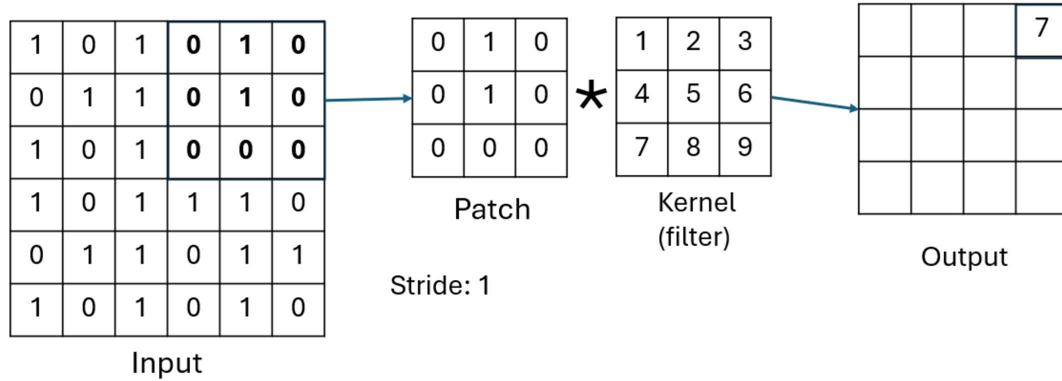

Figure 2.3: Convolution layer computation

convolutional filter, or kernel, is a smaller matrix that slides over the input matrix. As the kernel moves over the input matrix, it performs a dot product between the kernel and the patch of the input matrix that it currently covers. This dot product produces a single value, which is then stored in the output matrix. The kernel slides over the input matrix according to the stride parameter, which determines how many steps the kernel moves each time.

Hyperparameters such as the number of filters, stride, and padding influence the dimensions of the output data from that layer. The number of filters determines how many kernels are applied to the input matrix, with each filter producing an output matrix, thereby increasing the depth of the output. A higher stride means that the kernel steps over more fields, resulting in a reduced size of the output matrix. Padding involves adding extra fields around the border of the input matrix.

After each convolution operation, an activation function is applied. A popular one is Rectified Linear Unit (ReLU) introducing nonlinearity into the network. The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2.5}$$

where x is the input to the activation function. The ReLU function replaces all negative values with zero and leaves positive values unchanged.

Pooling Layer

Pooling layers (or downsampling layers) perform dimensionality reduction, reducing the number of parameters. To do so, an aggregation function as a filter is sweep across the input. The two main types of pooling are *max pooling*, which selects the maximum value in each receptive field, and *average pooling*, which calculates the average value. Pooling helps in reducing overfitting. An example of max pooling is listed in the figure 2.4

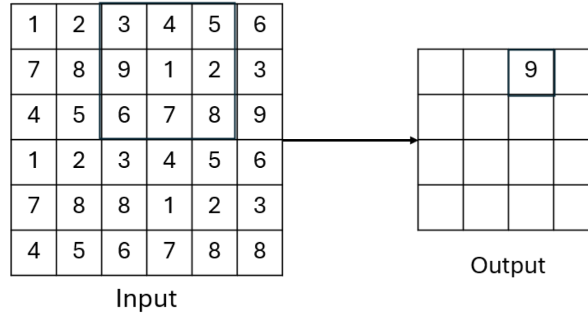

Figure 2.4: Max-pooling layer computation

In the example 2.4 provided, it is illustrated how performing max pooling on a 6x6 matrix with a stride of 1 affects the output dimensions, resulting in 4x4. By examining a specific patch of the matrix, the result is determined by selecting the maximum value within that patch.

Fully-Connected Layer

Classification takes place in the FC layer. Every node is connected to every node in the previous layer. This layer aggregates the features extracted by previous layers to classify the features into different categories. Typically, a softmax activation function is used in the final FC layer to output probabilities, indicating the likelihood of each class.

The Softmax activation function is commonly used in the final layer of a neural network for classification tasks. It transforms the raw output scores, also known as logits, into a probability distribution over classes. The Softmax function is defined as:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.6}$$

where z_i represents the score (or logit) for class i, and the denominator is the sum of the exponentiated scores for all classes j. This normalization ensures that

the outputs are in the range $[0, 1]$ and sum up to 1, making them interpretable as probabilities.

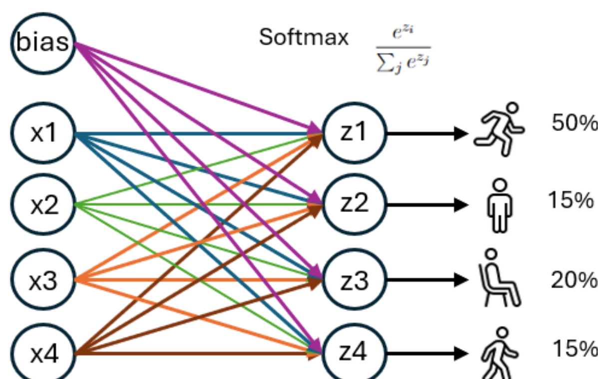

Figure 2.5: Softmax example

Figure 2.5 illustrates an example where the final layer of neurons applies the softmax function, ultimately producing the output as a probability distribution across the classes.

2.1.8 Activity Classification

Finally, to ensure the ML model can effectively interpret categorical activity labels, techniques such as one-hot encoding are employed. In particular, one-hot encoding transforms each activity into a binary vector, where each position in the vector corresponds to a unique category. The position corresponding to the activity is set to 1, while all other positions are set to 0.

Finally, to ensure the ML model interprets an activity defined as a categorical activity label techniques such as one-hot encoding are used. It involves mapping the activity into a binary vector where the position corresponding position is set to 1, and all other positions are set to 0.

For instance, in this study, four activities were considered: Walking, Running, Sitting, and Standing. These activities are encoded as follows:

- "Walking" → [1, 0, 0, 0]
- "Running" → [0, 1, 0, 0]
- "Sitting" → [0, 0, 1, 0]
- "Standing" → [0, 0, 0, 1]

2.2 Introduction to Federated Learning

ML models face a significant challenge: they often require vast amounts of data to perform effectively. Centralizing this data can present issues, particularly regarding data security and privacy. To address this, FL, also known as collaborative learning, offers a decentralized approach.

The concept was first proposed by Google in 2016 [18]. The basic idea is that the computation must be moved to the data. The main participants in the technique are the clients (data owners) and the server (responsible for aggregating and managing the model). The training process is organized into iterations called "rounds." During each round, the server setup a global model initializing the model parameters randomly or from saved ones, the model is sent to the connected devices, clients nodes perform local training on their data, and send the updates back to the server. Finally, the server aggregates these parameters using techniques like FedAvg, which computes a weighted average of the model weights received from clients. The weight of the average is determined by the number of samples each client used for training, ensuring that each data point has an equal influence on the global model.

Mathematically, it is expressed as:

$$\theta_{t+1} = \sum_{i=1}^{N} \frac{n_i}{n} \theta_i^t \tag{2.7}$$

where:

- θ_{t+1} is the updated global model at round $t+1$,
- θ_i^t represents the local model parameters from client i at round t,
- n_i is the number of data points on client i,
- $n = \sum_{i=1}^{N} n_i$ is the total number of data points across all clients.

To optimize the local models, common optimization algorithms employed include Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam).

FL allows for faster convergence since it leverages diverse datasets distributed across multiple clients.

2.2.1 Flower Framework

Flower is a notable open-source framework for FL, offering an extensive setup and practical examples for integrating FL with various ML frameworks. It is supported by a robust community of developers and data scientists, and includes code examples for deployment on edge devices such as Android and iOS. [18]

Flower is designed to be scalable, supporting large numbers of clients, and is adaptable to different client configurations. It is client-agnostic, accommodating various programming languages and hardware; communication-agnostic, allowing diverse data serialization and communication methods; and privacy-agnostic, supporting multiple privacy-preserving techniques. This versatility enables Flower to incorporate new FL algorithms and adjust to the evolving ML landscape.

The architecture of Flower is divided into two main components:

- **Server Side**: This includes the Client Manager, which manages ClientProxy objects and coordinates communication with clients; the FL Loop, which oversees the FL process; and the Strategy component, which configures training rounds, aggregates results, and supports different FL algorithms.
- **Client Side**: Consists of either an Edge Client Engine for real devices or a Virtual Client Engine for simulations. Both types of clients interact with the server through a protocol designed for model training and evaluation on local data partitions.

Flower separates the FL process from decision-making. The FL Loop handles the learning process, while the Strategy component makes decisions regarding FL procedures. This separation facilitates support for a range of client platforms and implementations, enhancing Flower's flexibility.

For communication, Flower employs bi-directional gRPC streams. gRPC was selected for its efficient binary serialization and its capability to handle multiple messages without additional connection overhead. This choice streamlines communication between clients and servers, minimizing manual configuration needs. Below is a brief overview of the gRPC protocol.

2.2.2 gRPC

Remote Procedure Call (RPC) is a protocol that enables a program to execute a procedure on a different address space (heterogeneous) as if it were a local call.

The process is abstracted so that the client code doesn't need to worry about the complexities of network communication. To enable procedure calls in a distributed environment, the following must be considered:

- **Server Location:** Dynamically locate the server before issuing a call.
- **Call Synchronization:** Manage and synchronize multiple concurrent calls on the server.
- **Argument Passing:** Address the challenge of passing arguments by reference due to disjoint address spaces.

- **Partial Failures:** Anticipate and handle potential network or server failures to ensure reliability.

The client sends a request to a server process that is listening for remote calls. This request includes the name of the procedure to be called and the parameters required. The server executes the procedure and sends the result back to the client. Procedure arguments and return values must be converted (marshalling/unmarshalling).

RPC can operate over various protocols such as HTTP, TCP, or UDP, which handle the data exchange between client and server.

Among RPC frameworks, **gRPC** is one of the most popular. Created by Google in 2016, gRPC is a modern, high-performance, open-source framework designed to simplify the development of distributed systems. It enables seamless communication between services, regardless of their underlying languages or platforms.

gRPC leverages HTTP/2 as its transport protocol, abstracting away the complexities of the protocol from developers. This allows them to work with a straightforward API without needing to manage the details of HTTP/2. gRPC uses Protocol Buffers (Protobuf) as its interface description language (IDL) to define the structure of data and services. Protobuf is a language-neutral, platform-neutral, binary serialization format that specifies the messages exchanged between gRPC clients and servers. The Protobuf compiler generates stubs, which serve as simplified interface for client to make remote calls handling serialization of request and deserialization of server responses.

With its efficient use of HTTP/2 and Protobufs, gRPC offers superior performance in terms of speed and bandwidth usage compared to many other RPC frameworks. gRPC's support for multiple programming languages and platforms makes it ideal for diverse environments.

gRPC is designed to handle high-performance needs in large-scale distributed systems, making it applicable for microservices architectures, cloud-native applications, and IoT.

2.3 Introduction to Transfer Learning

2.3.1 Tensorflow

Tensorflow (TF) is an open-source platform developed by the Google Brain Team under the Apache Open Source License, widely used for ML and DL applications. Released in 2015, TF was initially created to advance research in machine intelligence and deepen the understanding of deep learning. Google chose to make it open-source to accelerate AI development, which has led to its adoption by a large and active community of data scientists and developers. This broad support has made TF one of the most popular and widely used ML frameworks.[19]

TF can run on various computational platforms, including CPUs, GPUs, and Google's TPUs, without requiring code changes. This capability allows TF to be deployed across a wide range of devices, from portable devices to high-end servers.

Users can build models using data flow graphs within the library, where nodes represent mathematical operations and edges signify data represented by means of tensors: multidimensional array. Built on top of TF, Keras is a high-level API that makes deep learning model easier to be used. Offering smaller and more readable code, developers option this library because it requires less cognitve load.

2.3.2 TF Lite

Tensorflow Lite (TFLite) is an open-source deep learning framework designed for on-device inference, commonly referred to as Edge Computing. It provides both prebuilt and customizable execution environments, enabling the deployment of ML models on mobile, embedded, and IoT devices. TFLite is optimized for resource-constrained edge devices and supports multiple platforms, including Android, iOS, embedded Linux, and microcontroller units (MCUs). [20]

TFLite offers APIs that allow developers to generate and deploy optimized TF models on mobile and embedded devices. These models are compressed and fine-tuned to ensure high performance even on devices with limited computational resources. Additionally, TFLite uses FlatBuffers for data serialization and access.

2.3.3 Post-Training Quantization

For small devices with limited hardware capabilities, TF Lite offers quantization methods that reduce model size and efficiency, particularly by improving CPU and hardware accelerator performance while minimizing the impact on accuracy. The available options for post-training quantization are:

- **No Quantization:** During the conversion to TF Lite, compatibility issues might lead to conversion errors, resulting in unexpected results. Running the model without quantization can help verify that the original TF model's operators are compatible with TFLite. It also serves as a baseline for debugging errors introduced by subsequent quantization methods. For example, if a quantized model produces incorrect results but the floating-point model is accurate, the issue likely stems from the quantized version of the TFLite operators.

- **Dynamic Range Quantization (DRQ):** This method reduces memory usage and increases computational speed without requiring a representative dataset for calibration. It statically quantizes only the weights from floating-point to 8-bit integers at conversion time, achieving 8-bit precision.

Dynamic-range operators further optimize latency by dynamically quantizing activations based on their range, performing computations with 8-bit weights and activations. Although the outputs are stored in floating-point format, this method provides near-fixed-point inference latency, balancing performance with simplicity.

- **Full Integer Quantization (FIQ):** This approach provides further latency reductions, decreased memory usage, and compatibility with integer-only hardware accelerators by quantizing all model math to integer values. Full integer quantization requires calibration or estimation of the range (min, max) for all floating-point tensors in the model. Since variable tensors (such as model input, activations, and model output) need to be calibrated, the converter requires a representative dataset. This dataset typically consists of a small subset of few hundred samples of training or validation data.

Chapter 3

Related Work

Inspired by advancements in ML, a multitude of reviews and surveys have examined HAR and associated techniques using smartphones.

3.0.1 HAR in smartphone devices:

One of the pioneering studies in HAR using smartphone sensors was conducted by Saponas et al. [1] introducing iLearn, a system using the iPhone's accelerometer and Nike+iPod Sport Kit for real-time activity classification. By employing a Naïve Bayesian Network, iLearn achieves accurate recognition without retraining data in case of new users. However, this research necessitates active data collection from multiple devices in the early stage, followed by merging them for model training.

Surveys over the existing work related to HAR has been conducted. For example, Lara et al. [9] reviewed 28 HAR systems using wearable sensors, highlighting key design considerations. The study identified challenges such as learning approach, obtrusiveness, flexibility, recognition accuracy, and other important design issues. From the survey and many others like [12, 21] it is possible to outline many of the factors to consider when performing HAR.

Sampling rate, for example, Chung et al.[22] recommends using a low sampling rate, such as 10 Hz, which can effectively recognize activities while conserving battery life. However, other research suggests that higher sampling rates, typically between 20 Hz and 50 Hz, are better for capturing detailed activity patterns [23, 24].

Another factor to consider is the time sliding window size. Researches made on this matter [25, 9, 26, 27] discusses the varies number of seconds to consider. Sousa et al. [25] conclude that the optimal window to consider depends on the specific activity to be recognized and the attributes considered.

Gu et al. [21] emphasize the need for normalizing among data sources. While *min-max normalization* is common [28, 21, 29], it can be problematic in presence of

outliers, so *standardization* is often recommended. Proper data rescaling improves computational efficiency and model performance [30].

Dataset selection is a crucial factor in activity recognition research, particularly since many studies are conducted in non-laboratory settings, which can compromise the accuracy of results [31]. Samples from few partecipants per dataset can also hinder the generalizability of findings. When choosing a dataset, it is essential to consider the sensors employed, the context in which the data was collected, and the specific features included in the dataset. When dealing with imbalance datasets, study reports that to address dataset imbalance, *F1-score* as performance metric, is effective in imbalanced classification contexts [32].

Finally, sensor attributes and their placement are considerations to make in activity recognition research [33, 34]. Utilizing a combination of sensors rather than relying on a single sensor can enhance performance [35, 33]. The most commonly used sensors in the literature include accelerometers, gyroscopes, and, occasionally, magnetometers [36].

Over the past decade, common models for HAR include decision trees, Support Vector Machines (SVM), Naive Bayes, and Gaussian Mixture Models (GMM) [25]. Recently, DL models like CNNs and Recurrent Neural Networks (RNNs) have emerged as promising alternatives [28]. While CNNs excel in processing image data, recent studies have also applied them to sensor data, such as accelerometer readings, demonstrating their usage in HAR tasks [21, 29].

TL has found utility across various domains, including *text sentiment classification* [37], *image classification* [38], and HAR [39].

3.0.2 Federated Learning:

So far, the discussion has focused on methodologies for implementing HAR without addressing privacy concerns related to sensor data.

In the context of smartphone health applications, Majumder et al. [7] highlight the lack of focus on privacy and security in many publications on the area of the study, stressing the urgent need for robust data privacy-aware algorithms.

FL is a Google's technique [8], which adheres to data minimization principles [6], ensuring that only essential no sensitive information are collected on server. Researches based on this techniques have been conducted.

For example, a FL-based study [40] implemented a semi-supervised framework for segmenting COVID-19 affected regions in 3D chest CT scans from institutions in China, Italy, and Japan, ensuring patient data privacy. Accurately identifying anatomical structures and reducing false positives, it was generalized thanks to its across diverse datasets, including those without COVID-19 cases.

Additionally, research [41] presents the first commercial results of FL in training RNN language models for next-word prediction in virtual keyboards on smartphones.

The study compares server-based training using stochastic gradient descent (SGD) with on-device training utilizing the FedAvg algorithm, allowing multiple clients to collaboratively train a shared model without disclosing their raw data. The results demonstrate that the federated model achieves a 5% relative improvement in top-1 recall compared to the server-trained model when evaluated on client cache data, significantly improving keyboard prediction for applications like Gboard.

Another study on HAR in smart environments compared the use of local, centralized, and FL approaches, using RNNs. The results showed that both federated and centralized learning achieved better performance, with FL being the preferred option due to its enhanced privacy protection[5].

Finally, the article done by Beutel et al. [18] proposes the Flower framework effectively supports large-scale heterogeneous FL workloads, showcasing its potential across various applications, including healthcare, due to its strong privacy-preserving features.

3.0.3 Transfer Learning

To address all the above issues fasting the training time on a constrained device TL is introduced. TL reduces memory and computational requirements by minimizing the amount of training data needed, as compared to training models from scratch without pre-trained weights [42]. This has led researchers to focus on applying large pre-trained models to both similar and diverse tasks, aiming for broader adaptability and improved performance across different domains.

A study on vulnerable road user detection demonstrated that TL can enhance accuracy while reducing training time, achieving over 98.71% accuracy with smartphone sensor data [43]. To consider both storage efficiency and processing speed, the compression of CNN models is considered.

Studies have explored various techniques for model compression, demonstrating significant improvements in performance while maintaining accuracy.

Cheng et al. [44] examine quantization techniques and demonstrate that mobile image classification can achieve a 4-6× speedup and 15-20× compression while maintaining comparable accuracy.

Finally, Orășan et al. [45] investigated post-training quantization (PTQ) techniques in TensorFlow Lite to reduce model size by employing 8-bit integer representations instead of 32-bit floating-point numbers. Their study, analyzing small, medium, and large CNN models, highlights variations in compression ratios and accuracy changes based on different quantization methods.

3.0.4 Federated Transfer Learning (FTL)

Research combining previous main techniques represents a state-of-the-art advancement in the field, as there have been limited studies conducted on this topic across domains.

Chen et al. [46] present FedHealth, a FTL framework designed to enhance customization based on user data and facilitate data aggregation while preserving privacy evaluated in the contexts of HAR and Parkinson's disease.

Wang et al. introduce FTL-CDP [47], a FTL framework designed for cross-domain prediction in smart manufacturing. This framework allows groups of smart devices to share knowledge through a central server making it suitable for integration with existing smart manufacturing systems.

Chapter 4

System

4.1 System Overview

The architecture of the PHAR system is designed around a client-server model that facilitates efficient and scalable FL workflows. At its core, the system consists of an android client and a server that work together to manage and execute ML tasks.

The figure 4.1 shows the FedAvg algorithm, which enables multiple clients to collaboratively train a shared model without revealing their raw sensor data. In this implementation, clients maintain models that are partially pre-trained, meaning the base model section's weights are fixed, while the head model weights are updated using the generalized model's weights received from the server.

The communication between the client and the server is performed through the use of the *Flower* framework. Flower is a comprehensive framework that simplifies the complexities of FL communication. It automates much of the communication processes required for FL, including tasks such as model updates and aggregation. This automation is necessary for maintaining synchronization between multiple clients and the central server.

Subsequent sections of this document will provide a detailed examination of the specific components within this architecture and their interactions, explaining how they contribute to the system's functionality.

4.2 PHAR Android Application

The *PHAR Client* is an Android application developed for testing and evaluating TFLite models using techniques of TL and FL.

Developed using *Android Studio* and implemented in *Java* (version 1.8). Central to the application's functionality is the *Transfer API* library (version 2.16.1), which facilitates the ML operations necessary for both local and FL. This library provides

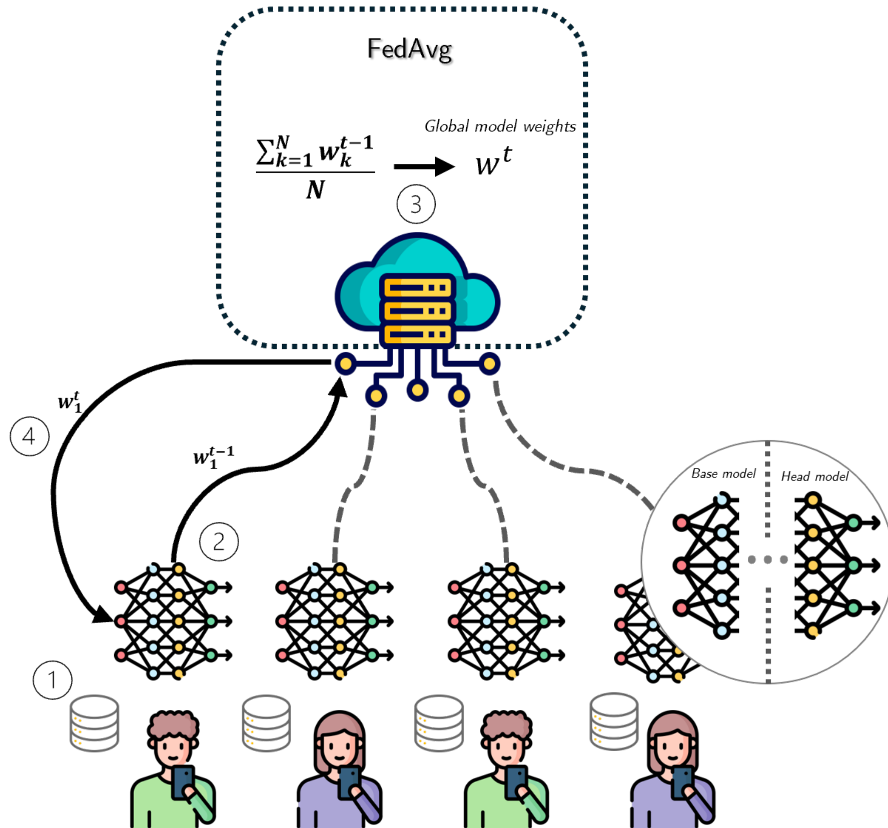

Figure 4.1: System overview

the tools needed to run TensorFlow Lite models.

To improve the user experience, the PHAR Client integrates a sophisticated notification system. This feature is designed to keep users informed about the ongoing status of tests, whether they are being conducted locally on the device or through FL with the server. Users are continuously updated on the progress of their tasks and are able to respond promptly to any issues that arise.

When tests demand significant storage resources, there's a risk of causing application crashes. To mitigate this, the notification system alerts users to potential issues. Indeed if a crash occurs, the system automatically restarts and resumes the tests from where it was interrupted. Users are notified if a problem arises, indicated by the disappearance of the notification, which signals that the testing thread has been suspended.

Managing non-garbage collected memory, allocated outside the typical heap managed by the garbage collector, was a major challenge in developing the application. This was particularly crucial when handling the large continuous binary buffers needed to store samples for training and testing, which presented unique

issues. Direct buffers were advantageous in this context, as they enable faster I/O operations by bypassing the need to copy data between the Java heap and native memory, while also reducing the strain on garbage collection processes. However, the trade-off is that direct memory requires careful manual management, which introduces a higher risk of memory leaks or crashes if not managed properly. The rapid allocation and deallocation of this memory compounded these risks, requiring substantial effort to ensure both the notification system and the overall memory management were combined to meet these demands without compromising application progress.

The application requires specific permissions to access device storage and data. These permissions are necessary for several functions, including running background threads, posting notifications, enable the FL connection, and storing test results in CSV format. The permissions requested include:

- **Access to network state** (`ACCESS_NETWORK_STATE`) to check the status of network connectivity.
- **Foreground service** (`FOREGROUND_SERVICE`) for running tasks in the foreground, even when the app is not actively used.
- **Receive boot completed** (`RECEIVE_BOOT_COMPLETED`) to ensure the app can start services after the device restarts. THis ensure that the device can restart testing in case of any crash.
- **Retrieve tasks** (`RETRIEVE_TASKS`) to access information about ongoing tasks.
- **Ignore battery optimizations** (`REQUEST_IGNORE_BATTERY_OPTIMIZATIONS`) to prevent the app from being affected by system battery-saving features.
- **Internet access** (`INTERNET`) to enable network communication.
- **Read and write external storage** (`READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`, `MANAGE_EXTERNAL_STORAGE`) for accessing and managing files on external storage. These are necessary to write, read the csv files stored in the device.
- **Data sync for foreground services** (`FOREGROUND_SERVICE_DATA_SYNC`) to designate that the ‘FlowerWorker‘ operates as a foreground service specifically for data synchronization tasks. This allows the service to transfer data between a device and the cloud over a network.
- **Post notifications** (`POST_NOTIFICATIONS`) for sending notifications to the user. These include both the local testing and federated testing.

- **Bind job service** (`BIND_JOB_SERVICE`) to allow the 'FlowerWorker' class to schedule and manage background tasks using the Job Scheduler. This permission ensures that only authorized components can bind to and execute jobs through the service, enabling management of long-running tasks such as server communication and data processing while conserving system resources.

4.2.1 Project overview

The *PHAR Client* is customizable to run with loaded pre-trained models, with options for quantization, and percentage of non-trainable layers. In the current version, the models consist of five trainable layers, and users can load the model with the specific number of trainable layers, ranging from 0% (fully trainable) to 83% (all layers pre-trained except for the final dense layer used for activity classification). Utilizing pre-existing knowledge from the pre-trained models and adapting them to new datasets decreases the chances of overfitting or consuming excessive computational power.

PHAR Client's features include:

- **Real-Time Sensor Monitoring:** Provides live monitoring of sensor data collected from the user's device. This feature ensures that the sensor data is valid and helps to identify any potential connectivity or accuracy issues before model training and evaluation begin.
- **Pretrained Model and Dataset Selection:** Users can choose from various pre-loaded models and datasets, with the option to work with quantized. Additionally, users can configure the number of trainable layers to suit specific testing scenarios.
- **Local Training:** In addition to supporting pre-trained models, the PHAR Client allows for local training and evaluation using datasets stored on the device. Users can adjust the trainability of model layers.
- **Federated Learning Testing:** The client supports FL, enabling collaborative model training across multiple devices without the need for centralized data sharing. This is especially useful for privacy-sensitive applications where data cannot be directly transferred between devices.

The application starts with a general description of the project.

4.2.2 Real-Time Sensor Monitoring

The Real-Time Sensor Monitoring feature provides a dynamic display of sensor data, enabling users to verify sensor readings as they are collected. This feature is

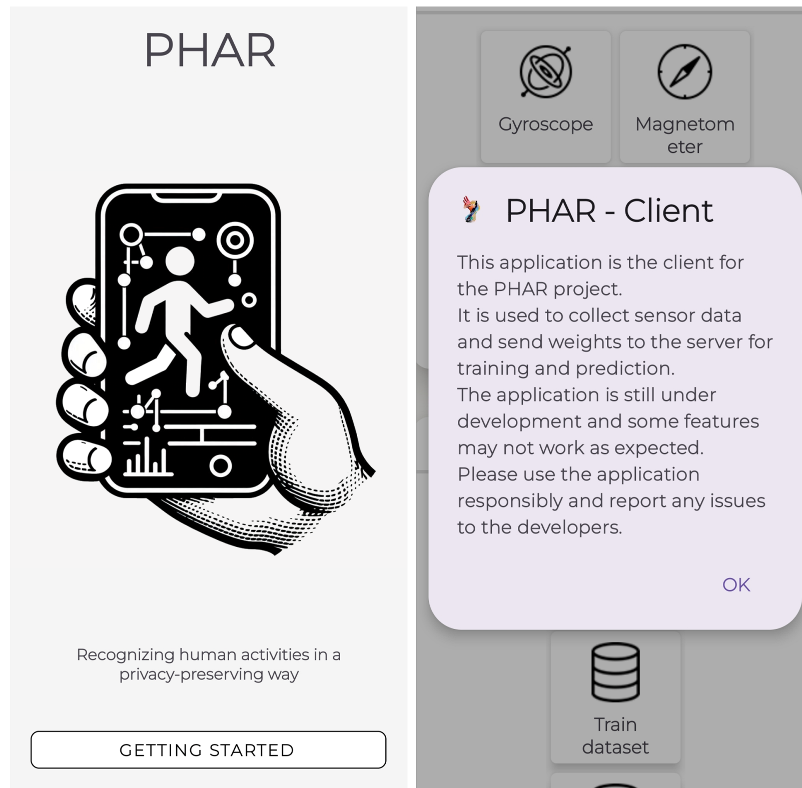

Figure 4.2: Overview

useful to ensuring that the application effectively captures real-time data, reflecting the actual sensor inputs.

Upon application startup, a 'SensorManager' instance registers listeners for the available sensors, including the gyroscope, accelerometer, magnetometer, and rotation vector sensors. The data from these sensors is stored in a matrix designed to hold up to 50 records, each containing multidimensional sensor inputs. This matrix enable tracking and recording the most recent sensor data.

As the sensors generate data, the 'SensorHandler' class processes this information in real time. The data is continuously updated in the matrix, and once the matrix reaches its capacity of 50 records, the data is saved to 'readyRecord'. This record can then be utilized for real-time training or analysis.

This feature ensures that sensor data is monitored, managed, and displayed in real time, providing accurate and timely feedback on sensor status and data collection.

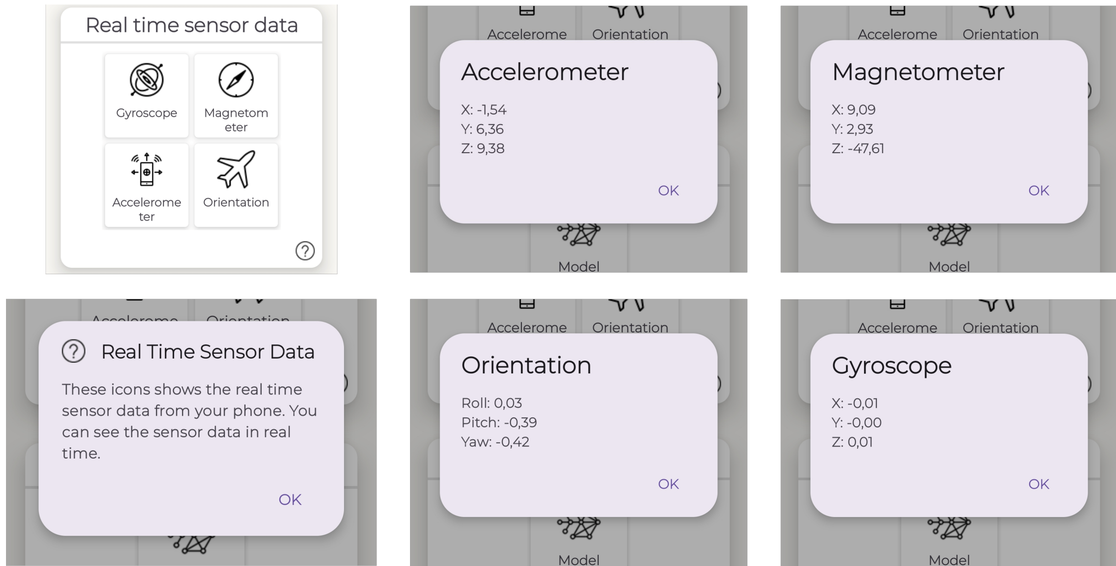

Figure 4.3: Real-Time sensor monitoring section

4.2.3 Test Configuration

This feature enables users to select from the available ML models within the application. By recursively scanning the model folder, it displays a nested directory structure that reflects the path to each model, along with its associated configurations. Current options include model quantization and the number of frozen pre-trained layers.

Users can also choose between different datasets for training and testing. The available datasets are *MotionSense* (MS) and *MobiAct* (MA), with the first dataset partition selected by default. Additionally, partitions can be selected for FL setup.

4.2.4 Local Testing

This section details the use of TL without FL. The current implementation provides a `Train and Test` button, which allows the system to test all available model and dataset combinations. To enhance the user experience, a real-time notification system updates users on the status of ongoing tests.

When the “Train and Test” button is selected, the system initiates a structured training and evaluation process. This is managed by the `TransferLearningLocal` class and the `FlowerClient` class. The process begins by iteratively selecting various parameters, such as the model type, total number of samples, and dataset combinations. The models differ in their quantization levels and layer configurations, offering a wide range of configurations for testing. The datasets considered include options such as *MS* and *MA*. The selected samples are divided into two parts: 80%

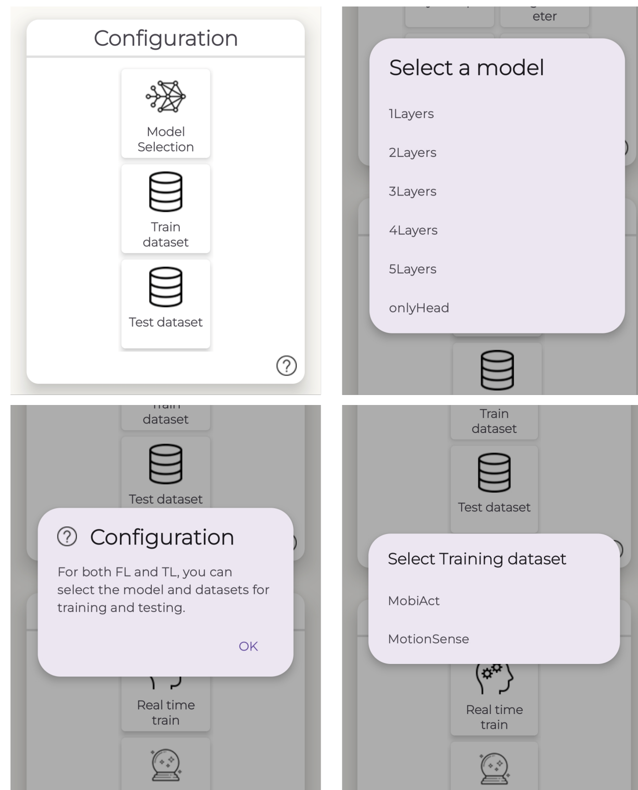

Figure 4.4: Test configuration overview

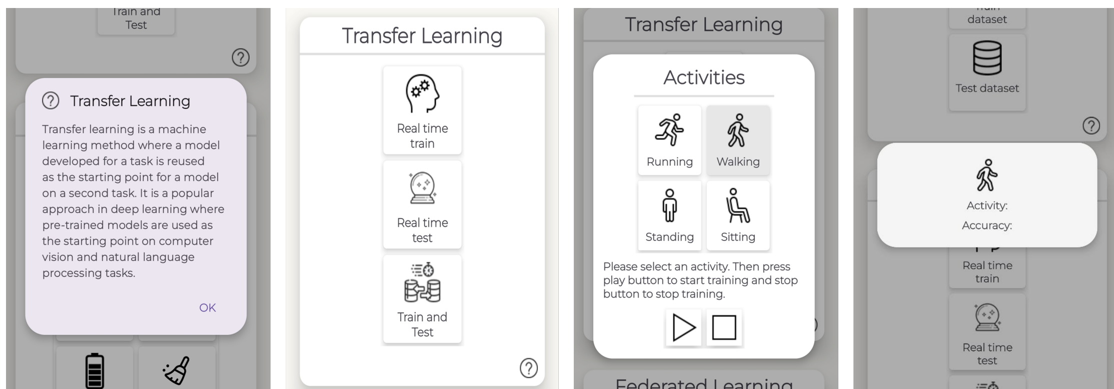

Figure 4.5: Transfer Learning Overview

of the data is allocated for training, while the remaining 20% is used for evaluation. When using a single dataset, this 80-20 split is maintained. In scenarios involving two datasets, 80% of the data is taken from the first dataset for training, while the remaining 20% is sourced from the second dataset for evaluation.

The datasets used have already undergone pre-processing. This includes normalization, the removal of outliers, and the synchronization of sampling frequencies, ensuring that the data is ready for the next stage. Each dataset is split into four portions, simplifying data selection for federated learning. The `NormalizedPreprocessing` class handles the extraction of samples and splits them into time-series sequences. Subsequently, these sequences are divided into smaller segments, referred to as "sections," which serve as input data for the model.

The method `createTimeSeries()` constructs time-series data by iterating over different activities (e.g., walking, jogging) and reading data from corresponding CSV files. The data is then structured into sequences with labels indicating the activity. Each sequence is comprised of multiple features, such as roll, pitch, yaw, and rotation rates, forming the input data for training and evaluation.

The `timeSeriesToSection()` method refines the time-series data by dividing it into smaller windows using a sliding window technique. Each window, or segment, contains a fixed number of samples (e.g., 50), and the sliding window moves with a predetermined step size (e.g., 10 samples), creating overlapping segments. This method is crucial in capturing both short-term and long-term temporal dependencies in the data. The resulting sections are structured so that 80% are used for training, while the remaining 20% are reserved for evaluation.

Each segment is shaped as a 9×50 matrix, representing 50 time steps for each of the 9 sensor dimensions. This ensures that for every sensor dimension, 50 consecutive records are preserved per segment, providing a comprehensive view of temporal patterns in the input data.

The `TransferLearningLocal` class manages the workflow, including the creation of notifications to indicate progress. It initializes a `FlowerClient` instance with the appropriate model path, sets up the model, and proceeds with training and evaluation.

The `FlowerClient` class is integral to this process. It acts as a wrapper around a federated learning model managed by the `TransferLearningModelWrapper`. This class handles model parameters and the training process. Specifically, the `FlowerClient` updates the model parameters, performs training, and evaluates performance. Training is conducted over a specified number of epochs, during which the model's weights are adjusted based on the provided data. Evaluation is performed by assessing the model's performance on the testing set, with metrics such as accuracy, loss, and F1-score being computed and recorded.

Once training and evaluation are complete, the results are saved in a CSV file for further analysis. This file includes metrics such as accuracy, loss, F1-score, and training time, providing a view of the model's performance across different configurations.

Although the real-time training and evaluation buttons are operative, they are not yet fully finalized. As a result, when users interact with these features, a toast

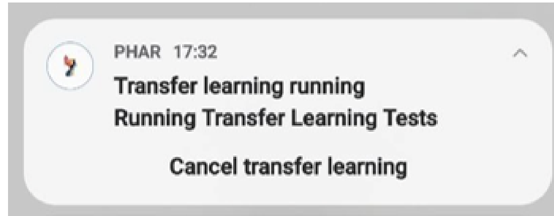

Figure 4.6: Transfer Learning notification

notification appears, displaying the message "coming soon". This indicates that while initial testing has been completed successfully, further refinements are still in progress.

The *Real-time Train* button, when clicked, presents an Alert Dialog that lists the currently available activities. Users, by selecting one of the displayed activities, can initiate the recording process. The system waits for the sensor data matrix to update, after which it moves the data to the model input. Once the user is satisfied with the training for the selected activity, they can terminate the session by pressing the easily recognizable stop button.

On the other hand, the *Real-time Test* button provides a different function. When clicked, it displays an Alert Dialog that predicts the most probable activity the user is engaged in, along with the corresponding accuracy. This gives the user immediate feedback on the model's performance based on the current sensor inputs.

4.2.5 Federated Testing

At the bottom of the application page, users have can configure and manage settings pertinent to FL tasks. Users can enter the server's IP address, port number, dataset partition, and model details. After inputting this information, clicking the `Start Client` button initiates a validation process to ensure the accuracy of these settings. Upon successful validation, the client establishes a gRPC connection with the server, which then confirms the connection.

Once the connection is established, the Flower worker begins its operation by opening a gRPC channel to receive messages from the server. This operation is handled by a dedicated executor thread, which prevents the main thread from being blocked and ensures that the application continues to function smoothly.

Additionally, a notification like the one in Figure 4.7 is displayed. It indicates that FL is currently running. The user has the option to cancel FL if desired.

The `FlowerWorker` class plays a central role in managing the FL tasks. It facilitates communication with the server through gRPC, handling various messages that may include requests for model parameters, training instructions, and evaluation commands. The class's `connect()` method establishes the gRPC connection with

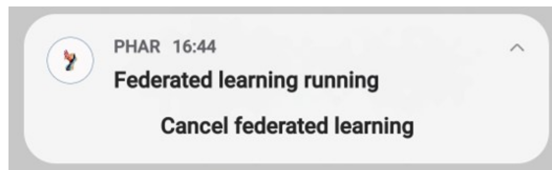

Figure 4.7: Federated Learning Notification

the server using the provided IP address and port. This connection is crucial for ongoing communication throughout the FL process.

When messages are received from the server, the `FlowerWorker` processes them appropriately. For instance, the methods `handleFitIns()` and `handleEvalIns()` are responsible for managing the fitting and evaluation processes. These methods parse the received data, perform the necessary computations, and prepare responses to send back to the server.

To maintain application responsiveness, the `FlowerWorker` utilizes an executor thread to handle gRPC communication. This setup ensures that the main thread remains unblocked. Additionally, the worker is configured to run in the foreground to prevent the system from terminating it during long-running tasks. This is achieved by creating a persistent notification through the `createForegroundInfo()` method.

Robust logging mechanisms is also integrated into the worker. If an error occurs or if the task is canceled, detailed logs are recorded in the "FlowerResults.txt" file. These logs capture the status and outcomes of operations.

Users can disconnect from the network at any time by clicking the `Stop Client` button. This action activates the worker's `onStopped()` method, which sets a cancellation flag, notifies observers that the worker has been stopped, and sends a cancellation message via gRPC. As a result, an error is raised, indicating that the client has left the network.

Furthermore, a `Clear Logs` button allows users to remove any unnecessary logs related to the connection history. These logs, which are stored in the "Flower-Results.txt" file, include details about the connection status and FL operations, complete with timestamps for each action. They also document training and evaluation metrics such as accuracy, loss, and training time.

4.2.6 TensorFlow Library

Once the Flower worker is initialized, it sets up the necessary components for the transfer learning model. This involves loading various models: the initialization model, bottleneck model, training head model, inference model, and optimizer model. Each model serves a specific purpose in the training and inference process.

To process dataset samples, the `TransferLearningModel` class converts the

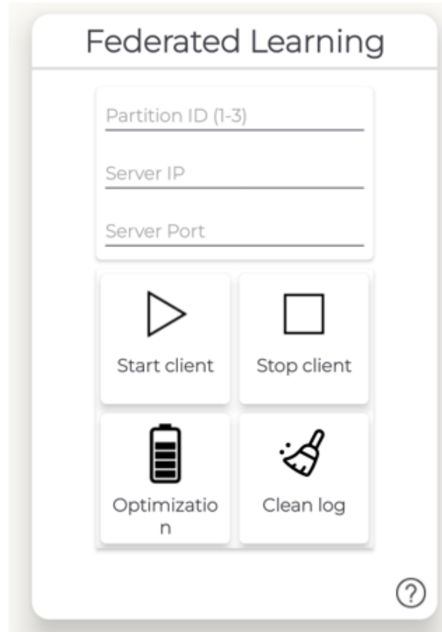

Figure 4.8: Federated Learning Overview

input data into a format suitable for the model. This conversion process is handled in several steps:

- **Conversion to DirectByteBuffer:** Input data, initially represented as a 2D float array (`float[][]`), is transformed into a `DirectByteBuffer`. Each element of the array is converted to bytes and placed into the `DirectByteBuffer`.

- **Bottleneck Generation:** The `DirectByteBuffer` containing the input data is passed through the base model, where the bottleneck representation is generated. The output of the base model is referred as the bottleneck.

- **Sample Storage:** The generated bottleneck, along with the associated activity label, is stored in either the training or testing samples list, depending on the sample's purpose. This ensures proper categorization for model evaluation and training.

During training, the model updates its parameters using the stored training samples. It processes data in batches, computes gradients, and performs optimization steps to enhance performance. Testing involves evaluating the model on the test samples to measure accuracy and other performance metrics, comparing predictions with activity labels to assess effectiveness.

The `TransferLearningModel` class is central to the TL process, managing the lifecycle of the model, including initialization, training, inference, and parameter updates.

The `LiteBottleneckModel` class generates bottleneck representations from input data using a TensorFlow Lite model. It processes data to extract feature vectors efficiently.

The `LiteInferenceModel` class manages the inference process, utilizing bottleneck representations and model parameters to produce predictions.

The `LiteTrainHeadModel` class focuses on the training head of the model, updating parameters based on computed gradients.

The `LiteOptimizerModel` class handles optimization, updating the optimizer state and applying algorithms to enhance model performance.

The `LiteInitializeModel` class sets initial parameter values for the model before training begins.

To generate these files being loaded in the android application it is necessary to use a TFLite convertor. It consists of generation of two sequential neural network inserting the layers respectively for the base model and the head model. In the first, pre-trained weights generated from the entire MS dataset, avoiding random initialization and providing a solid foundation for training. The base model is saved in TensorFlow's SavedModel format, necessary for conversion to TensorFlow Lite. The head model, responsible for classification and on-device training, is defined to process from the base model.

The final step involves using the `TFLiteTransferConverter` class to convert the combined base and head models into TensorFlow Lite format. This class manages the conversion, generating models for initialization, bottleneck processing, training, and inference. It also offers options for quantization to optimize the model. The resulting TensorFlow Lite model is then saved.

The figures below show examples of the generated converted files. In particular the one shown is a CNN model that uses representative dataset quantization and Adam optimization with a learning rate of 0.0001. The first three trainable layers are frozen.

- **Bottleneck Model:** Figure 11 shows the compressed feature vector representation of the input data.
- **Training Head Model:** Figure 14 illustrates the model's head used for classification tasks.
- **Optimizer Model:** Figure 12 depicts the model's optimizer used for updating parameters.
- **Initialization Model:** Figure 10 demonstrates the initial setup of model parameters.

- **Inference Model:** Figure 13 shows the model used for generating predictions.

4.3 Flower Learning Server Setup

The server configuration begins by defining the training and testing datasets, selecting the model, and determining the number of sections for the current trial. After loading the model and freezing its base weights, the server is initialized with the `FedAvgAndroid` strategy, a customized version of the Federated Averaging (FedAvg) algorithm. This strategy coordinates the distributed learning process by handling client sampling, averaging local updates to refine the global model, and optionally including custom functions for configuring training and evaluation.

Additionally, the client setup involves initializing clients with the model, datasets, and configuration settings like the maximum number of sections to use. The server also tracks cumulative training time, updating it at each round by collecting the maximum training time from clients and adding it to the cumulative total.

The strategy sets parameters such as the minimum number of clients required for training and evaluation, the fraction of clients involved, and acceptable failure flag. It also defines an evaluation phase, where the global model is assessed using metrics like loss, accuracy, and F1 score, with results saved in CSV format after the final round. The server aggregates and updates the global model by reconstructing weights based on updates from clients.

Performance evaluation occurs after each round of training, with metrics aggregated to provide insights into the model's effectiveness. This ongoing evaluation guides adjustments for subsequent rounds, ensuring progress in the FL process.

4.3.1 How Flower Server and Clients Collaborate

The interaction between the Flower server and its clients follows a well-defined sequence, coordinating federated learning steps between the server and participating clients. The following steps outline the process of how the Flower framework manages the training and evaluation of models in a federated setting:

1. Initialization of Model Parameters: The process begins with the Flower server obtaining the initial model parameters. These parameters, typically representing the global model, are initialized using the `initialize_parameters` method. The initialized parameters are then sent to the clients, enabling them to begin the federated training process.

2. Configuration of Federated Training: Once the initial model parameters are shared with the clients, the server invokes the `configure_fit` function to configure the next round of federated training. The `configure_fit` method generates a list of client proxies and their respective `FitIns` (Fit Instructions),

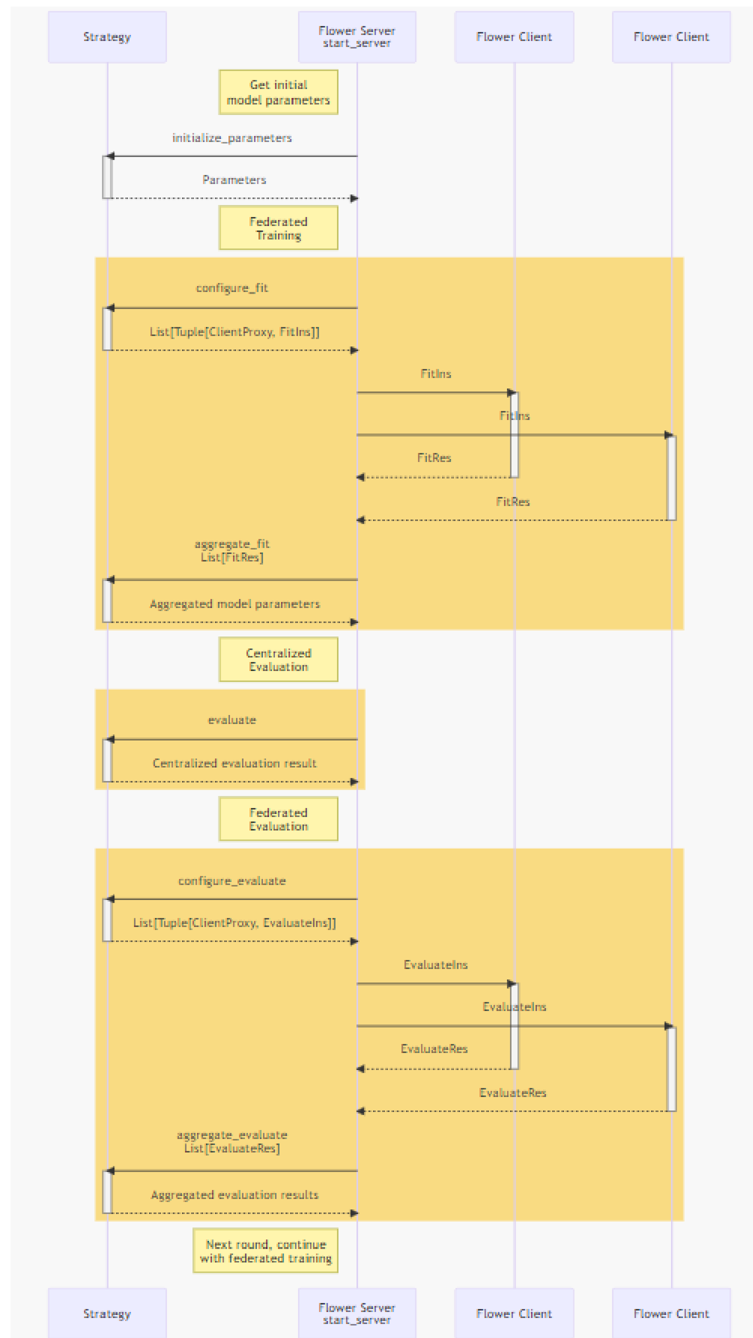

Figure 4.9: Overview of the Flower Server and client Interaction from [48]

which consist of the current global model parameters and configuration settings. These instructions are sent to the selected clients.

3. Local Training on Clients: Each participating client receives the `FitIns`

and proceeds to perform local training on its private dataset. The clients complete the training process and return the results, encapsulated in `FitRes` (Fit Results), to the Flower server.

4. Aggregation of Model Parameters: The server then aggregates the locally updated model parameters received from the clients using a weighted averaging approach. This aggregation process generates a new set of global model parameters, reflecting the contributions of each client. The aggregated parameters are sent back to the clients, completing one round of FL.

5. Centralized Evaluation: The customized Flower server perform centralized evaluation using the updated global model. The server can evaluate the model's performance using a validation dataset partition available to the server.

6. Configuration of Federated Evaluation: Next, the server initiates the evaluation phase by invoking the `configure_evaluate` method. This function generates evaluation instructions (`EvaluateIns`) for the clients, containing the global model parameters and any necessary configuration. The instructions are sent to the selected clients.

7. Local Evaluation on Clients: The clients evaluate the global model using their local validation data and return the results in the form of `EvaluateRes` (Evaluation Results).

8. Aggregation of Evaluation Results: The server aggregates the evaluation results received from the clients, allowing it to assess the global model's performance based on the client data. The aggregated evaluation results are used to determine the effectiveness of the model after the current round of FL.

9. Iterative Rounds of Training: This process repeats iteratively, with the server between rounds of federated training and evaluation until the pre-configured number of rounds is completed.

Chapter 5

Experimental results

In this section, we delve into the core analysis conducted for the thesis, outlining the actions taken, the results obtained, and the insights gained from the tests performed. The results were derived from a series of experiments designed to validate initial assumptions, with each test serving to confirm or challenge the hypotheses put forward.

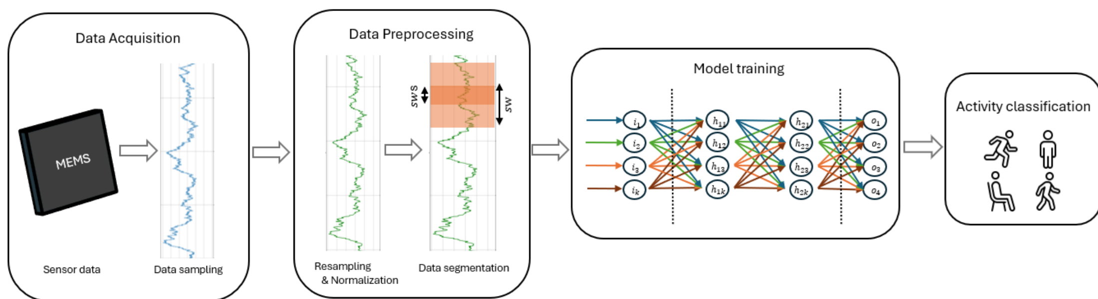

Figure 5.1: Activity recognition process

The figure 5.1 shows the process of activity classification using sensor data and a deep neural network. It involves data acquisition, preprocessing, model training, and classification. The following sections highlight the results and process involved during this thesis research.

5.1 Data acquisition

To collect data for the experimental results, a review of widely used datasets was conducted, with a focus on those utilizing sensors commonly found in smartphones. Specifically, the combination of accelerometers, gyroscopes, and magnetometers was considered because capable to classify activities of daily living with high accuracy

(over 91%) as studies shows [25]. The accelerometer used to measure the 3D acceleration of the phone, the gyroscope to measure the 3D angular velocity and the magnetometer.

To achieve this, the Android Studio environment was initially explored to understand the main components and how to manage sensor interactions and activities. According to the Android documentation [49], these sensors have been used together since Android 2.3 (API Level 9), released in 2010, and are now supported by over 99.8% of Android devices [50].

This approach was selected with the aim of applying real-time sensor data from mobile devices in future applications. Based on this objective, the MS and MA datasets were chosen for analysis. Table 5.1 highlights key aspects of these datasets, focusing on the portions relevant to this research.

Dataset	Records	Classes	Participants	Sampling Frequency	Sensors
MS	1,412,865	6	24	50 Hz	A, G, M
MA	16,756,325	20	67	200 Hz	A, G

Table 5.1: Overview of the datasets. A: Accelerometer, G: Gyroscope, M: Magnetometer

MS: Data were collected using an iPhone 6s placed in the participant's front pocket, utilizing the accelerometer and gyroscope at a 50 Hz sampling rate. During the experiment, participants wore flat shoes and placed the smartphone in their front trouser pocket. The dataset comprises trials from 24 participants performing various activities, including walking, jogging, sitting, standing, and upstairs and downstairs . Data collection was supervised, with participants actively labeling the activities.

Class	Samples	Percentage
JOG	134,231	9.50%
SIT	338,778	23.98%
STD	306,427	21.69%
WLK	344,288	24.37%
Total	**1,123,724**	**100.00%**

Table 5.2: Class distribution of samples in the MS dataset.

MA: Data were collected using a Samsung Galaxy S3 smartphone equipped with the LSM330DLC inertial module, which includes accelerometer, gyroscope, and orientation sensors. Sampling was performed at the highest possible rate using the "SENSOR_DELAY_FASTEST" setting (200 Hz). The smartphone was placed in the participant's trouser pocket, with data recorded freely, without any fixed orientation, to simulate everyday use. The dataset includes recordings from 57 participants performing activities including walking, jogging, upstairs, downstairs,

sitting, standing. Data collection was supervised.

For this analysis, the selected activities are from the ambulation group: sitting, standing, walking, and running. The chosen attributes include acceleration (both linear and gravitational), the rotation rate, and the smartphone's orientation (attitude).

Class	Samples	Percentage
JOG	362,039	4.30%
SIT	226,492	2.65%
STD	3,592,445	42.48%
WAL	3,650,240	50.57%
Total	**7,831,216**	**100.00%**

Table 5.3: Class distribution of samples in the MA dataset.

The class distribution is imbalanced, making the F1-Score a useful metric in the last section[32]. Indeed, the F1 score is a useful metric for measuring the performance of classification models when you have imbalanced data because it takes into account the types of errors—false positives and false negatives—and not just the number of predictions that were incorrect.

5.2 Data preprocessing

5.2.1 Resampling

Studies have shown that adopting a sampling rate lower than 200Hz not only increases the sparsity of activity data but also effectively captures abrupt motions characteristic of primary activities like the previously discussed. The optimal range for sampling such activities is between 20Hz and 50Hz [27, 24]. Consequently, reducing the sampling frequency of the MA dataset is essential, not only to minimize data storage but also to ensure compatibility with other datasets used in this study.

To achieve this, a fourth-order Butterworth filter was employed to ensure a flat frequency response within the passband, preventing aliasing during the downsampling process. The resulting filter response 5.2 is shown below. After applying the filter, decimation was carried out as part of the down-sampling process by selecting 1 out of every 4 consecutive samples.

5.2.2 Normalization

After the resampling phase, it is necessary to rearrange the range of values. Normalization options were considered, taking into account factors such as the presence of outliers and the variance from the mean. A careful examination of the dataset

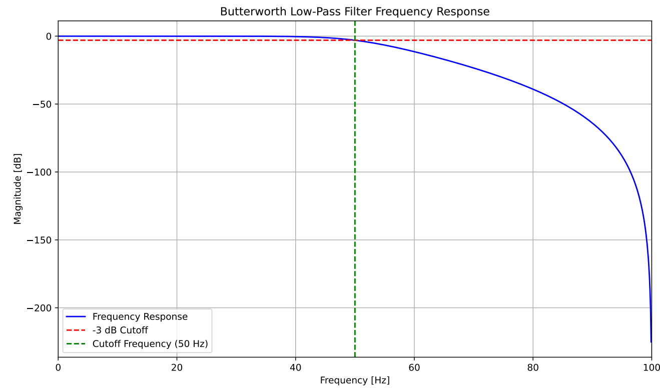

Figure 5.2: Frequency response of the Butterworth low-pass filter.

samples before normalizing the attributes revealed a Gaussian distribution in most of the attributes. In this context, applying Z-Score standardization could effectively remove outliers within the range of [-3, 3].

To give more insights of the value observed in the figures **?? ??**, the tables 5.4 intend to summarize the attributes statistics for the MS dataset and 5.5 intend to summarize the attributes statistics for MA one.

Attribute	**Mean**	**Std**	**Median**	**Min**	**Max**
$\text{attitude}_{\text{roll}}$	-0.18	1.57	0.12	-3.14	3.14
$\text{attitude}_{\text{pitch}}$	-0.99	0.49	-1.18	-1.57	1.57
$\text{attitude}_{\text{yaw}}$	-0.15	1.54	-0.14	-3.14	3.14
rotationRate_x	0.01	1.29	-0.00	-17.37	10.47
rotationRate_y	0.01	1.23	0.00	-18.41	17.54
rotationRate_z	0.01	0.81	0.00	-12.15	11.44
acceleration_x	0.04	0.45	0.02	-6.53	8.11
acceleration_y	0.80	0.63	0.88	-4.83	8.00
acceleration_z	-0.10	0.52	-0.04	-7.84	7.88

Table 5.4: Summary statistics of various attributes in the dataset before normalization.

The rationale behind this step is to prevent outliers from distorting model training, which could otherwise impact performance. Although removing these outliers might result in excluding some samples associated with highly intensive activities, such as running, efforts were made to minimize class imbalance. By defining a specific range, the aim is to limit the influence of outliers and focus on a more representative dataset, thus improving the quality and reliability of next model quantization and training.

Attribute	Mean	Std	Median	Min	Max
$\text{attitude}_{\text{azimuth}}$	181.54	107.05	192.62	-89.80	360.00
$\text{attitude}_{\text{pitch}}$	-71.42	57.01	-86.54	-180.00	180.00
$\text{attitude}_{\text{roll}}$	1.37	16.81	0.94	-88.14	89.95
acc_x	0.18	3.25	0.21	-19.44	19.53
acc_y	7.60	6.16	9.63	-19.56	19.59
acc_z	0.02	2.77	-0.22	-18.62	18.51
gyro_x	-0.02	1.03	-0.02	-10.00	9.97
gyro_y	-0.01	1.01	0.00	-10.01	10.01
gyro_z	0.01	0.63	0.01	-9.99	9.94

Table 5.5: Summary statistics of various attributes in the dataset MA before normalization.

To further validate this choice, evaluating four normalization functions would necessitate extensive testing of the normalized datasets across all combinations of hyper-parameter settings. For practice, it was decided to experiment with the Adam optimizer using a learning rate of 0.0001 and comparing the performance of the primary normalization functions as in figure 5.7. The figures show the results considering multiple partitions such as 2000, 2500, 3000, 4500 and 5000 samples, further discussion will be conducted on the following section reasoning them.

The results indicate that both Robust normalization and Z-Score standardization achieve the highest training accuracy. While Robust normalization shows a smoother curve with minimal variation across epochs, both methods perform similarly well on the testing dataset. To validate these findings, the two methods were further compared using the RMSProp and Adam optimizers, as shown in the figures 5.8, 5.9, 5.10, 5.11 for a detailed comparison.

In conclusion, Z-Score standardization was selected based on the outcomes of the previous analysis and discussion.

5.2.3 Data Segmentation

Dataset partitioning

This subsection addresses the next critical step required for both the Python and Android implementations. Prior to this, it is important to note that the thesis aims to employ FL in an environment with limited data storage. To accommodate this constraint and support the intended analysis, the data was divided into four partitions per dataset. The 5.6 intends to show the MS distribution after normalization and having removed the outliters. P: partition

The 5.7 intends to show the MA distribution after normalization and having removed the outliers.

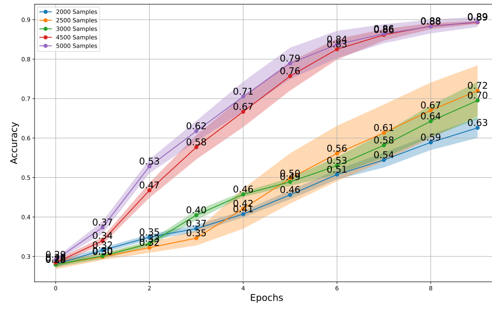

Figure 5.3: Impact of Range normalization on model accuracy

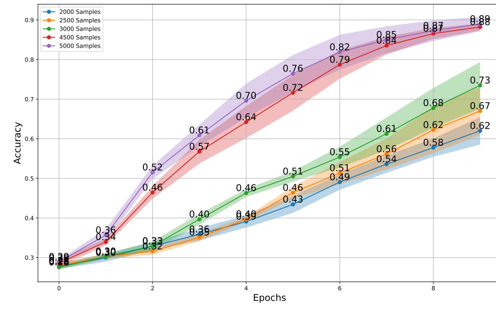

Figure 5.4: Impact of MinMax normalization on model accuracy

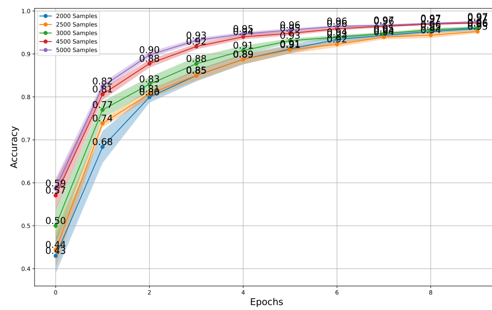

Figure 5.5: Impact of Robust normalization on model accuracy

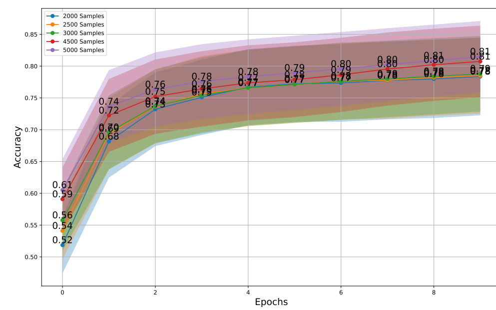

Figure 5.6: Impact of Z-Score standardization on model accuracy

Figure 5.7: Comparison of the effects of different normalization techniques on model accuracy over multiple samples settings.

Table 5.6: Class-wise sample counts and percentages per partition, including overall partition totals and their percentage of the entire MS. P: partition

P	JOG	SIT	STD	WLK	% of Dataset
P 1	20,709	84,693	75,804	78,388	25.00%
P 2	20,709	84,693	75,804	78,388	25.00%
P 3	20,709	84,693	75,803	78,388	25.00%
P 4	20,708	84,693	75,803	78,388	25.00%
Total	**82,835 (10.22%)**	**338,772 (41.78%)**	**303,214 (37.40%)**	**313,552 (38.61%)**	**100.00%**

Definition of the data segmentation

On the client side, data is accessed by partition, with a limit set on the maximum number of rows retrieved to minimize memory usage, particularly on Android devices. Despite this precaution, the application experienced significant memory allocation in a short period, leading to frequent activation of the garbage collector,

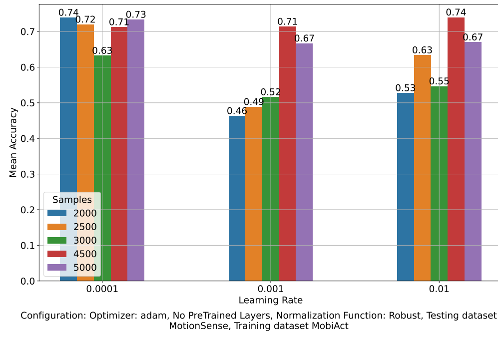

Figure 5.8: Effect of Robust normalization on model accuracy with Adam optimizer

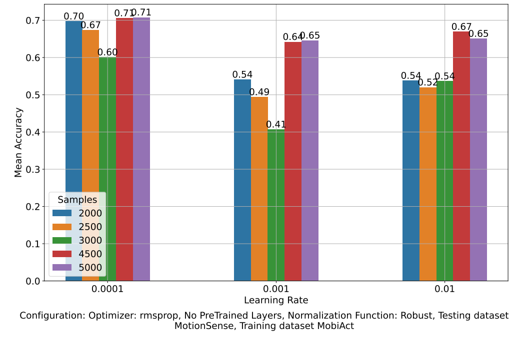

Figure 5.9: Effect of Robust normalization on model accuracy with RMSProp optimizer

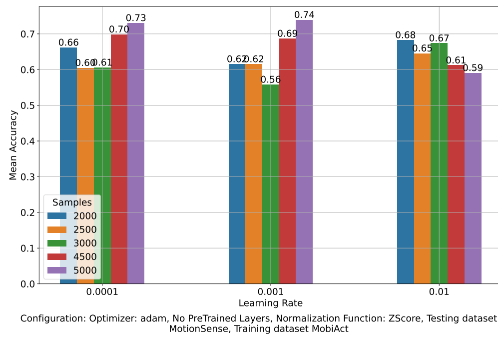

Figure 5.10: Effect of Z-Score standardization on model accuracy with Adam optimizer

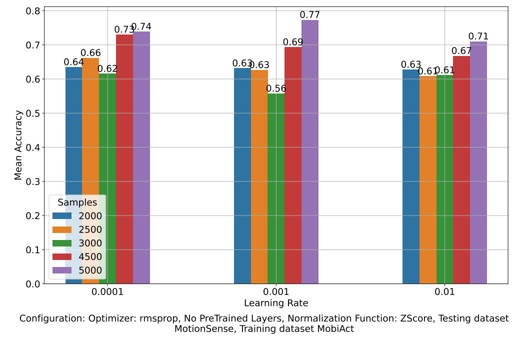

Figure 5.11: Effect of Z-Score standardization on model accuracy with RMSProp optimizer

Figure 5.12: Comparison of the effects of different normalization techniques on model accuracy across various sample settings and hyperparameters.

Table 5.7: Class-wise sample counts and percentages per partition, including overall partition totals and their percentage of the entire MA. P: partition

P	JOG	SIT	STD	WLK	% of Dataset
P 1	12,294	6,509	223,081	181,543	25.00%
P 2	12,293	6,509	223,081	181,543	25.00%
P 3	12,293	6,509	223,081	181,543	25.00%
P 4	12,293	6,509	223,080	181,543	25.00%
Total	**49,173 (2.90%)**	**26,036 (1.54%)**	**892,323 (52.68%)**	**726,172 (42.87%)**	**100.00%**

which negatively impacted performance.

Subsequently, the rows containing the attributes were combined with a Boolean vector representing activity labels, using a one-hot encoding scheme where a true value corresponds to the relevant row. The attributes of the time-series data were

then organized into sliding window vectors with with an overlap of 20% (a step size of 10). This configuration created a sliding window of 50 samples (equivalent to one second at 50 Hz) , continuing until the maximum number of sections was retrieved and ready for input into the model. To prevent class imbalance, data retrieval was halted once an equal number of instances from each class were stored, despite the significant class imbalance present in the original dataset.

5.3 Model

The model chosen is a customized version of the one proposed in [51]. The model showed in the figure 15 is a sequential CNN. The architecture consists of several convolutional layers, max-pooling layers, dropout layers, and FC layers to process the input data, followed by an activation function for classification. The input to the network is a tensor of shape (9, 50), which represents a 2D input with 9 rows, 50 columns. The output layer consists of 4 neurons, corresponding to the 4 possible classes for the classification task.

The activation function used is softmax, which converts the output into probability distributions over the classes.

The loss function applied during training is categorical cross-entropy, which is appropriate for multi-class classification tasks. For a multi-class classification problem, it is defined as

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(p_i) \tag{5.1}$$

where:

- C is the number of classes.
- y_i is a binary indicator (0 or 1) representing whether class i is the correct class.
- p_i is the predicted probability of class i.

Before implementing techniques that may impact performance, it is essential to evaluate the model's baseline performance.

Table 5.8: Baseline performance metrics for the MA-MA and MS-MS datasets.

Dataset	Accuracy	Loss	F1 Score	Training Time (s)
MA-MA	0.9969	0.0094	0.9969	1158.82
MS-MS	0.9958	0.0150	0.9958	824.93

Table 5.10 displays the results from training the model on 80% of the total samples, with 20% set aside for testing, thereby establishing a baseline for future experiments.

Next, we tested a combination of the two datasets. This approach maintained comparable results despite considering factors like outlier selection, normalization, and methodological considerations, as shown in Table 5.11.

Table 5.9: Resulting metrics for the combination of datasets

Dataset	**Accuracy**	**Loss**	**F1 Score**	**Training Time (s)**
MA-MS	0.9964	0.0105	0.9964	1179.90
MS-MA	0.9970	0.0110	0.9970	780.88

The impact of previously discussed factors on overall performance was minimal. A comparison of the model's performance on the Android and Python clients highlights the effects of model conversion and sample reduction, simulating real-world scenarios with storage constraints.

Figure 5.13 displays performance metrics for various training and testing sample configurations on both clients. It illustrates the impact of sample reduction on model performance, emphasizing the trade-offs associated with limited resources.

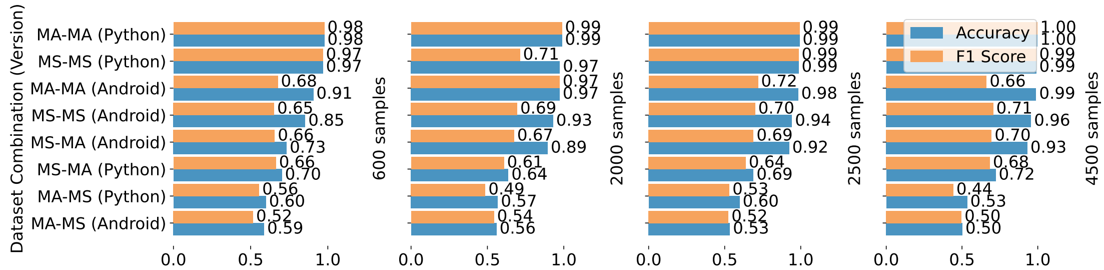

Figure 5.13: Comparison of model performance across different sample configurations and clients (Android vs. Python).

The accuracy drop is linked to dataset combination and class imbalance, as shown in Table 5.7. For instance, the 'SIT' class in the MA dataset comprises only 6,509 samples, representing just 1.54% of the total. This limited representation hinders the model's ability to generalize effectively, as achieving a balanced test scenario would necessitate 600 samples for each class, which is not feasible for our experiment.

To ensure clarity, a sample size of 2,500 has been selected for further analysis. More insights on different sample configurations can be found at [52].

Model conversion and implications

To adapt the model's architecture for deployment on Android, it is utilized the TF library for conversion to TFLite. This process involves splitting the model into two distinct components: a base model and a head model. The base model comprises frozen layers, which are kept untrainable during the conversion, while the head model retains its trainable properties. The frozen layers are preloaded with weights derived from a model previously trained on the MS dataset, which is substantial in size.

A significant challenge in this migration was ensuring compatibility between TF and TFLite during layer definition. Any incompatibility could lead to the removal of operations from the computational graph during conversion, resulting in unexpected behaviors or compilation errors. Therefore, meticulous attention to layer compatibility is was essential to maintain the model's integrity. TFLite provides limited support for TensorFlow operations, but some operations can still be processed by TFLite even without direct equivalents. For example, operations like tf.identity can be removed from the graph entirely, tf.placeholder can be replaced by tensors, and simpler operations such as tf.nn.bias_add can be fused into more complex ones. However, even certain supported operations may sometimes be removed during this process, necessitating close attention to layer compatibility to ensure a smooth conversion. Another important consideration was ensuring that the output of the base model did not require excessive storage, particularly when using direct buffers. Direct buffers allow memory allocation outside of the Java heap, optimizing both speed and preventing the garbage collector from managing this memory. While Android generally discourages the use of direct buffers, they become necessary when handling large datasets—such as during model training—to prevent unexpected deallocation of buffer input data.

A large base model output would impose significant computational and memory costs. For example, in the architecture described above, where the first convolutional layer is frozen and the subsequent layers are trainable, the output of the base model is (9, 46, 50). After generating the bottleneck (the output of the base model), this results in approximately 80 kB of data per processed input. In a test scenario with MAX_SAMPLE set to 4500, the device would need to store approximately 355 MB in continuous byte allocation. Such a large memory requirement could lead to the termination of the Android application due to excessive resource usage. To address this issue, it was necessary to deallocate all previous data after each test, ensuring that the minimum amount of memory was used at runtime. Even when using heap memory to store the bottleneck data, the garbage collector would automatically reclaim space for newly allocated data, potentially leading to inefficiencies. In contrast, utilizing direct memory buffers significantly reduces garbage collection overhead, thereby minimizing long pauses. This approach facilitates efficient I/O

operations by avoiding unnecessary data copies between the heap and native memory. It also surpasses heap size limitations, allowing for the handling of larger datasets.

For the seek of examples, it was considered for the test implementation to convert all possible configuration of the architectures. For the seek of exploring new concepts the converted models were further compacted by considering Post-training quantization.

5.3.1 Post-training quantization

Quantization refers to the process of reducing the storage size required for a machine learning model while maintaining its usability. This optimization not only minimizes the model's size but also enhances latency, thereby reducing computational delays with minimal impact on accuracy [45]. In this thesis, it was analyzed two specific quantization techniques: Dynamic Range Quantization (DRQ) and Full Integer Quantization (FIQ). Further details can be found in section 2.3.3. The memory usage of the quantized models is illustrated in Figure 5.14. This figure provides a visual comparison of how different quantization methods affect model size.

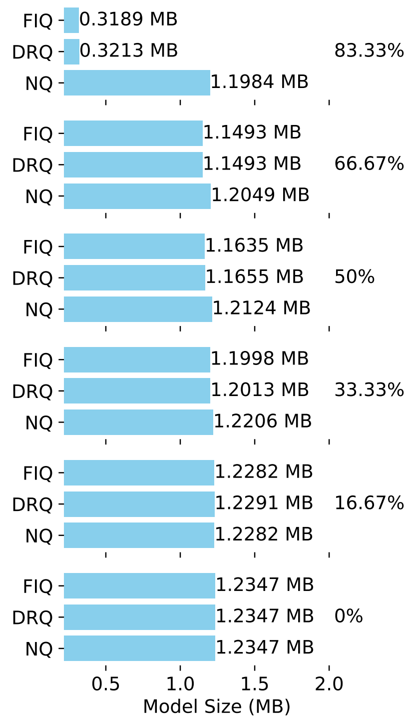

Figure 5.14: Effect of quantization on the model size

As illustrated in the figure 5.14, the effect of quantization is influenced by the

number of pretrained layers that are frozen. PTQ is indeed applied to the base model being converted, as it is not the model used for training in the subsequent phase. It is evident that a higher degree of quantization leads to a significant reduction in overall model size. As studies have also shown, the reduction of the overall compression size ratio tends to be lower [45] However, it is crucial to note that pretraining the entire model does not guarantee maintaining performance in terms of accuracy or F1-score. Adapting the model to new datasets may be essential for optimal results. This will be further clarified in the following section, where a detailed comparison of techniques is presented.

5.4 Classifications and Evaluations

The final stage of the HAR process involves analyzing the performance of the CNN with various configurations. The primary metric used for comparison is accuracy, calculated as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.2}$$

where:

- TP is the number of True Positives,
- TN is the number of True Negatives,
- FP is the number of False Positives,
- FN is the number of False Negatives.

In this context, accuracy is averaged across the four classes being predicted(5.2). Additionally, the loss, evaluated as categorical cross-entropy, is another considered metric (5.1). Training time is measured from the start of the first epoch to the end of the last epoch. Instead for FL, the cumulative training time is calculated as the sum of the time from the start of the first epoch round and the worst training time after each round. The formula for cumulative training time in FL is given by:

$$\text{Cumulative Training Time} = T_1 + \sum_{i=1}^{N} T_i \tag{5.3}$$

where:

- T_1 is the worst training time for the first epoch round,
- T_i is the worst training time for the i-th round,

- N is the total number of rounds.

T_x represents the maximum training time observed among the clients participating in round x. It is defined as:

$$T_x = \max(T_{x,1}, T_{x,2}, \ldots, T_{x,n}) \tag{5.4}$$

where $T_{x,i}$ denotes the training time of the i-th client in round x. This value provides an upper bound on the training duration for that round. Finally, the F1-Score is also used as a metric. It can be described as:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.5}$$

where

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN} \tag{5.6}$$

with:

- TP is the number of True Positives,
- FP is the number of False Positives,
- FN is the number of False Negatives.

Model analysis

Before testing any techniques that could introduce performance trade-offs, it's crucial to first assess the baseline performance of the model without any modifications.

Dataset	Accuracy	Loss	F1 Score	Training Time (s)
MA-MA	0.9969	0.0094	0.9969	1158.82
MS-MS	0.9958	0.0150	0.9958	824.93

Table 5.10: Baseline performance metrics for the MA-MA and MS-MS datasets.

Table 5.10 provides a summary of the results obtained from training the model on 80% of the dataset, with the remaining 20% reserved for testing. These initial results serve as an encouraging starting point for further experimentation.

Subsequently, a test was performed using a combination of the two datasets. By altering the source of data for both training and testing, this method highlights potential imperfections introduced by factors such as the selection of outliers, the normalization function applied, and previous methodological considerations. This step shown in the table 5.11 is critical for understanding the impact of these factors on the model's overall performance.

Table 5.11: Resulting metrics for the combination of datasets

Dataset	Accuracy	Loss	F1 Score	Training Time (s)
MA-MS	0.9964	0.0105	0.9964	1179.90
MS-MA	0.9970	0.0110	0.9970	780.88

Upon comparing the tables, it is evident that the results are comparable. The potential loss attributed to the factors previously discussed did not significantly impact the overall performance.

Sampling analysis

Subsequently, a comparison of the model deployed on the Android client versus the Python client highlights the impact of model conversion and the reduction of samples on both training and testing datasets. This aspect can be used to simulate real-world scenarios where storage constraints limit the ability to store and load the entire dataset for training and testing. As a result, the differences in performance between clients under varying sample configurations become apparent.

Figure 5.13 illustrates the performance metrics (e.g., accuracy and F1-score) of the model across different configurations of the training and testing datasets. These configurations involve both varying the number of samples and comparing the results from Android and Python clients. The figure highlights how reducing the number of available samples affects model performance, emphasizing the trade-offs involved when working with constrained resources.

The observed drop in accuracy can be attributed to the combination of datasets, where class imbalance plays a critical role. As highlighted in Table 5.7, the class distribution is uneven. For instance, when the MA dataset is partitioned, the 'SIT' class contains only 6,509 samples, accounting for a mere 1.54% of the total dataset. This class imbalance means that the model may struggle to generalize well, especially with fewer examples from the 'SIT' class.

To further analyze the impact of this imbalance, we calculate the number of time windows that can be extracted from the 'SIT' class, considering overlapping windows. The calculation is based on the following parameters:

- **Total number of samples**: 6,509
- **Time window size**: 50 samples
- **Overlap between consecutive windows**: 10 samples

The first step is to determine the **stride**, which represents the number of samples between the starting points of two consecutive windows. The stride is computed as:

$$\text{Stride} = \text{Window Size} - \text{Overlap} = 50 - 10 = 40$$

Next, the total number of windows can be determined using the following formula:

$$\text{Number of Windows} = \frac{\text{Total Samples} - \text{Window Size}}{\text{Stride}} + 1$$

Substituting the relevant values:

$$\text{Number of Windows} = \frac{6509 - 50}{40} + 1 = \frac{6459}{40} + 1 \approx 161.475 + 1 = 162$$

Thus, the number of time sliding windows that can be derived from the 'SIT' class, with the specified overlap, is approximately 162.

This relatively small number of windows underscores the challenges faced by the model in effectively learning from such a limited representation of the 'SIT' class, further contributing to the observed accuracy drop. For the sake of understanding, a test scenario with the appropriate class balance has been conducted with total samples of 600 as shown in the table 5.12

Table 5.12: Performance metrics for various dataset combinations (Samples: 600)

Dataset Combination	Version	Accuracy	Loss	F1 Score	Training Time
MS-MS	Python	0.977500	0.101616	0.977494	3.585166
MA-MA	Python	0.968333	0.198703	0.968030	3.446056
MA-MA	Android	0.905882	0.282084	0.676185	0.013107
MS-MS	Android	0.853782	0.519019	0.652068	0.000000
MA-MS	Python	0.702500	0.699651	0.663948	3.657154
MA-MS	Android	0.733333	0.489209	0.656030	0.000000
MS-MA	Python	0.587500	1.081621	0.515804	3.633167
MS-MA	Android	0.599167	0.912857	0.555218	0.013107

To maintain clarity and organization, a selection of 2,500 samples has been established. This approach facilitates the introduction of various configurations and constraints, enabling a clearer interpretation of the subsequent results. For further insights, testing performed across different sample configurations can be accessed at [52].

Transfer Learning Analysis

Figures 5.15 and 5.16 illustrates the relationship between accuracy and the number of pre-trained models for both the Android and Python versions.

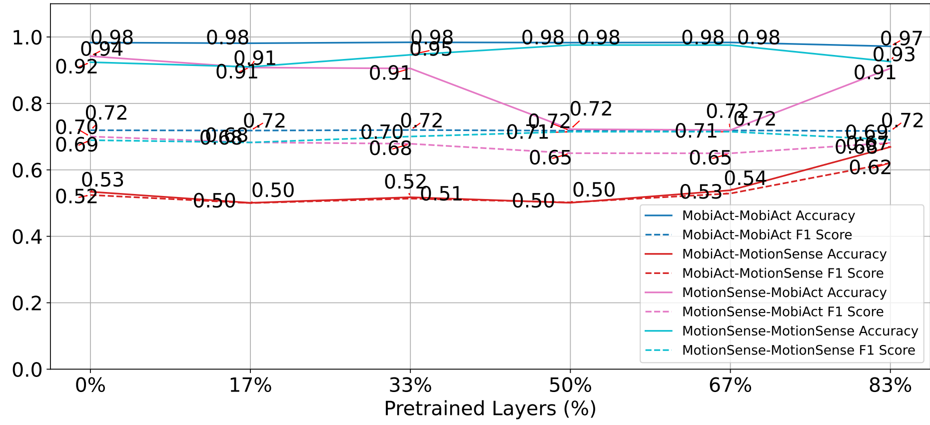

Figure 5.15: TL: Classification metrics vs. Number of Pre-trained Models for Android version

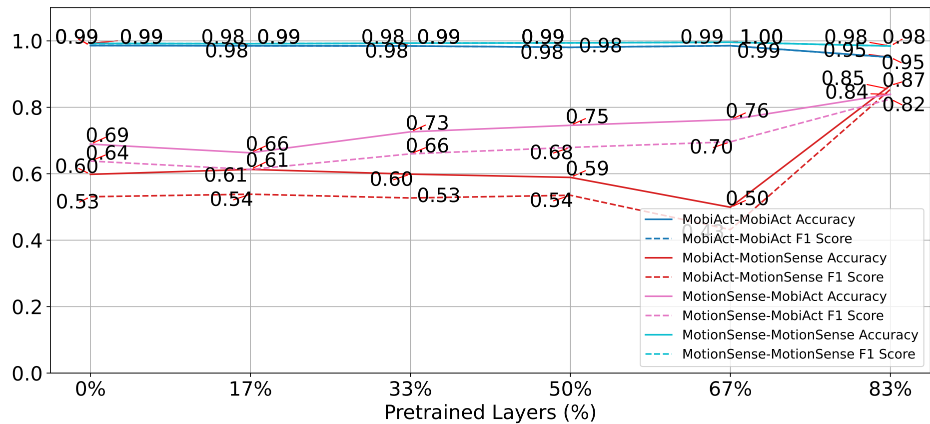

Figure 5.16: TL: Classification metrics vs. number of pre-trained models for python version

Figure 5.17 and 5.18 show that training time decreases as the number of pre-trained layers increases. Notably, there is a significant difference in training times among the Python and Android models, due to distinct implementations of the TF library and the computational limitations of Android devices.

These findings demonstrate the feasibility of training models on Android and provide a benchmark against the Python version.

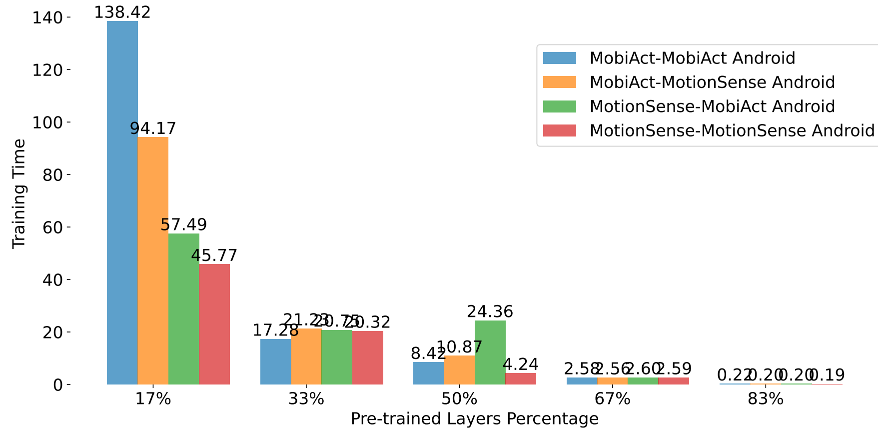

Figure 5.17: TL: Training time vs. number of pre-trained models for Android version

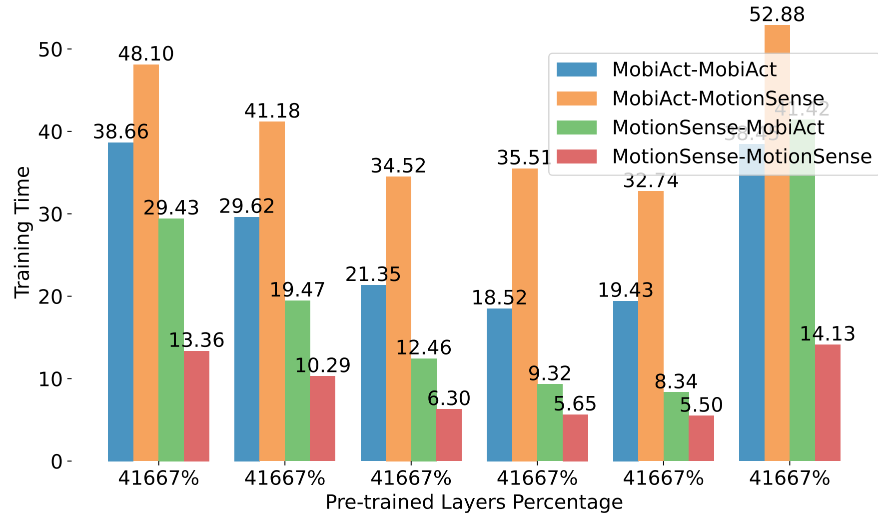

Figure 5.18: TL: Training time vs. number of pre-trained models for Python version

Quantization

To further analyse these data, considering quantization could save the size, as already seen, and training time at cost of accuracy. The bar plots in figure 5.19 shows these results.

Figure 5.21 shows that training time is affected by the quantization method. Training time decreases exponentially with increased quantization of early layers, while later layers exhibit similar times across methods.

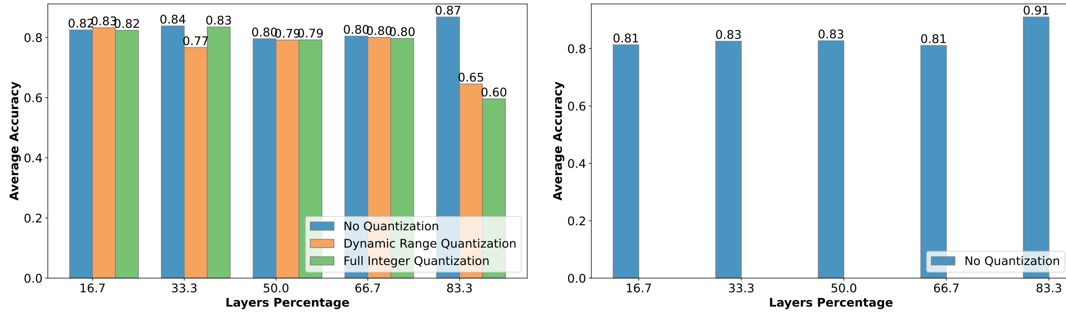

Figure 5.19: TL: Accuracy for quantization method averaged on dataset combinations using pre-trained layers

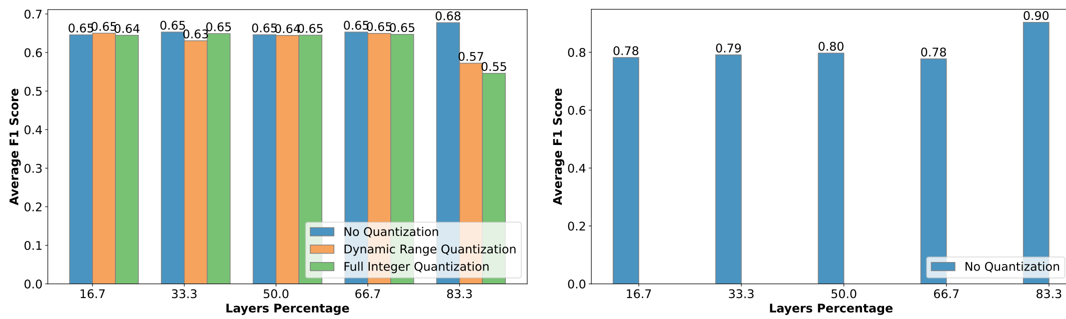

Figure 5.20: TL: F1-Score for quantization method averaged on dataset combinations using pre-trained layers

Though quantization generally has little effect on accuracy, results from the 83% pretrained layers model test reveal that combining it with additional datasets can complicate generalization to unseen data, influenced by both the quantization methods and the selection of pre-trained layers.

Federated Learning analysis

In this test benchmark, the first three dataset partitions were used for client training, while the final one was set aside for server-side testing. Training time is replaced by cumulative training time (see Equation 5.3) to account for FL iterations. It is crucial to identify trainable layers and deserialize weights before loading them into the server model for testing, as client-side training errors can affect model integrity.

The migration from local testing to FL began analyzing the Python implementation. The initial goal is to analyze the differences in training time between the TL and FLT approaches and subsequently evaluate the model performance in predicting human activities. Table 5.13 provides a comparative analysis of training

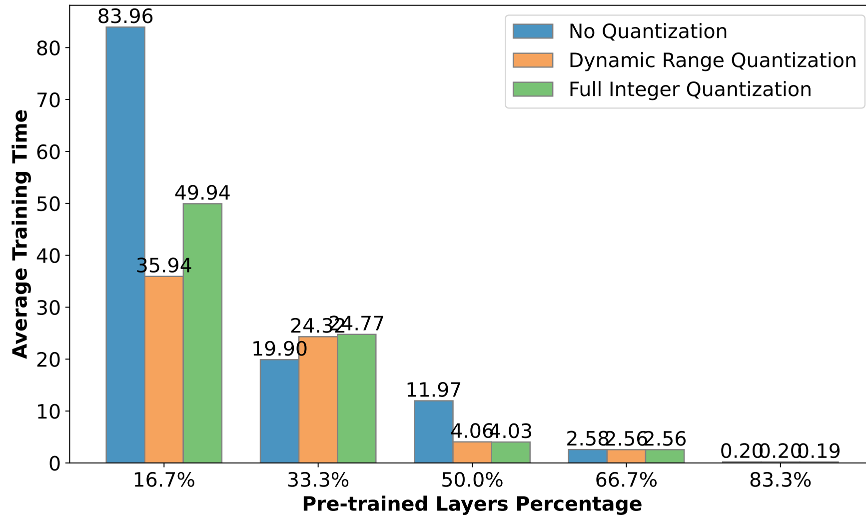

Figure 5.21: TL: Training time for quantization method averaged on dataset combinations using pre-trained layers for Android version.

and cumulative times for the Python implementation, emphasizing the percentage change in training time between the two approaches across models with pre-trained layers. This table highlights the performance of both techniques across all dataset partitions.

The analysis considers the ratio of total training time to the number of epochs, where, for FL, the effective epochs are calculated as # epochs × # rounds. The results underscore that the FL approach yields significantly faster responses during the training phase.

Table 5.13: Comparison of Training Time (TT) between No-FL and FL for Python version.

% Layers	No-FL TT (Mean)	FL TT (Mean)	% Change
0.00%	99.59	63.04	-36.54%
16.67%	86.88	50.85	-41.73%
33.33%	50.90	30.52	-40.00%
50.00%	37.10	22.79	-38.56%
66.67%	32.75	20.82	-36.18%
83.33%	29.45	19.51	-33.73%

It is important to emphasize that the same comparison could not be performed when transitioning from a single partition analysis to a full dataset analysis in the Android version. This is because loading the entire dataset into memory is not computationally feasible on a standard smartphone.

Transitioning from a local to a Federated Learning (FL) approach not only promotes privacy and collaboration among devices but also optimizes the utilization of computational resources. The findings reveal that these advantages are further enhanced by a reduction in overall training time. Performance metrics, including accuracy and F1-score (as shown in Table 5.13), indicate that the quality of the model remains largely unaffected.

Table 5.14: Comparison of Accuracy and F1-Score between No-FL and FL for Python version.

% Layers	Accuracy			F1-Score		
	No-FL	FL	% Change	No-FL	FL	% Change
0.00%	0.90	0.87	-3.47%	0.90	0.86	-5.23%
16.67%	0.89	0.84	-6.12%	0.88	0.81	-7.82%
33.33%	0.90	0.84	-6.37%	0.89	0.81	-8.78%
50.00%	0.90	0.85	-5.61%	0.90	0.82	-7.91%
66.67%	0.90	0.85	-5.46%	0.89	0.82	-7.63%
83.33%	0.91	0.89	-2.71%	0.90	0.88	-2.90%

Liming the samples to consider per test to 2,500, figures 5.22 and 5.23 illustrate the impact of varying the number of pre-trained models on accuracy and F1-score metrics in the Android and Python versions using FTL.

The plots highlight significant potential for enhancing generalization across datasets. Given that the pre-trained layers are derived from the MS dataset, the model demonstrates higher performance on MS-MS test sets. However, comparisons between the Python and Android versions are limited by their performance disparity; Python employs FedAvg while Android utilizes FedAvgAndroid, additionally different computational resources and training libraries are being used.

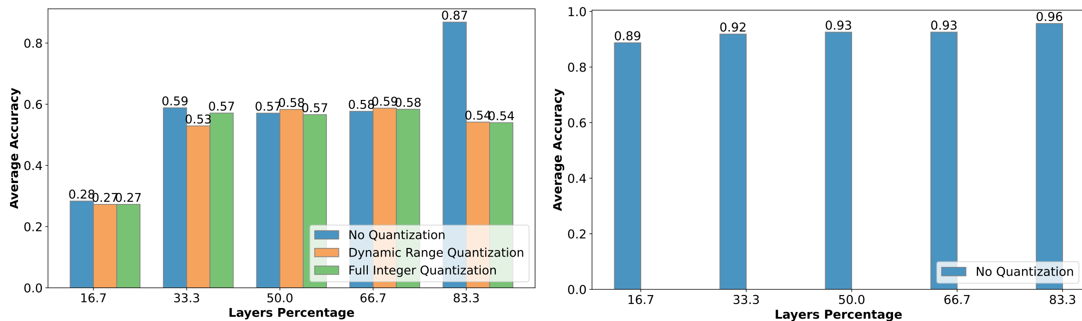

Figure 5.22: FTL: Comparison of accuracy for Android and Python versions

Ultimately, the integration of these techniques with FL, resulting in FTL, indicates a notable decline in performance metrics such as accuracy and F1-Score

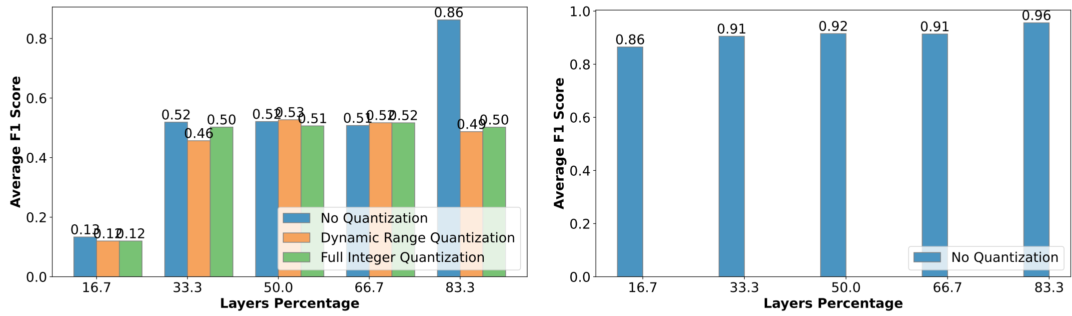

Figure 5.23: FTL: Comparison of f1-score for Android and Python versions

(see Figures 5.22 and 5.23). This suggests that the model may not yet generalize effectively across all dataset partitions. It's important to highlight that the previous analysis (Figure 5.19) focused solely on one of the four partitions, while this evaluation considers the entire dataset. Consequently, the average difference in F1-Score is approximately 23%, likely attributable to the initial dataset imbalance observed during the experiment.

Chapter 6

Conclusions

This thesis investigates and implements ML techniques to gain a comprehensive understanding of their application in smartphone-based systems for HAR. The contributions of this work are outlined as follows:

- **Development of PHAR**: A privacy-preserving smartphone application tailored for HAR, which integrates TL, FL, FTL, and PQT. This customized app serves as a valuable tool for in-depth analysis, effectively addressing the individual facets of TL, model compression, and FL.
- **Exploration of Techniques**: An examination of these techniques within the context of CNN, illustrating the incremental integration of each approach and tracking their progress and impact throughout the thesis.
- **Demonstrating Gains**: An analysis of the potential for privacy and training time improvements with a corresponding minimal reduction in model performance metrics (e.g., accuracy and F1-score) achieved through the application of FL on edge devices.
- **Addressing Deployment Challenges**: A focus on overcoming the challenges associated with deploying large-scale models on resource-constrained mobile devices while ensuring optimal model performance and scalability within the Flower framework.

The results indicate a possible reduction in training time of up to 41.73% when comparing no-FL and FL pretrained models, along with a 25% decrease in model size on mobile devices achieved through quantization, albeit at the cost of some efficiency in performance metrics.

Future research will focus on enhancing model generalization and enabling real-time prediction capabilities within the PHAR application. The scalable architecture

proposed in this work shows significant promise for continuous learning and live predictions. Additionally, further exploration of FL can be conducted by integrating alternative strategies such as FedAdagrad, FedAdam, FedSGD, and FedYogi.

This thesis contributes to the field by investigating FTL in the context of HAR and optimizing memory usage for resource-constrained devices, all while achieving substantial reductions in training times and model sizes.

Appendix

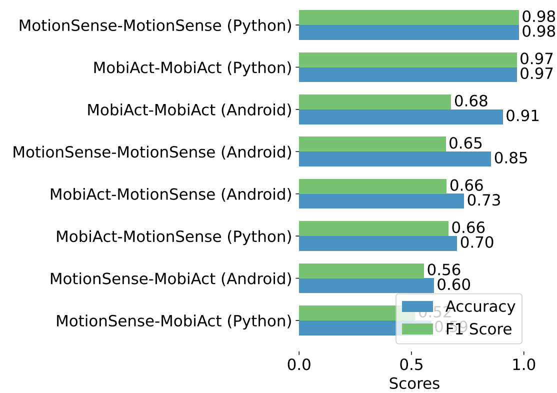

Figure 1: Performance of 600 samples configuration with no pretrained layers before FL-TL

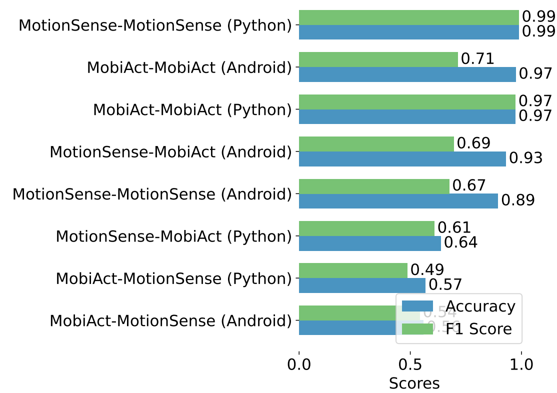

Figure 2: Performance of 2000 samples configuration with no pretrained layers before FL-TL

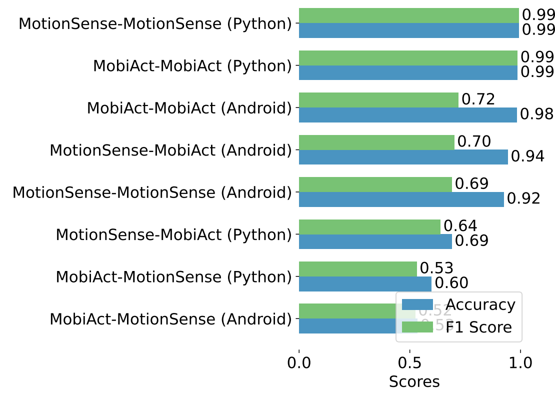

Figure 3: Performance of 2500 samples configuration with no pretrained layers before FL-TL

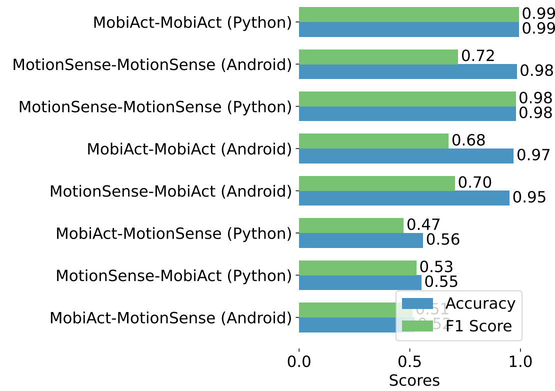

Figure 4: Performance of 3000 samples configuration with no pretrained layers

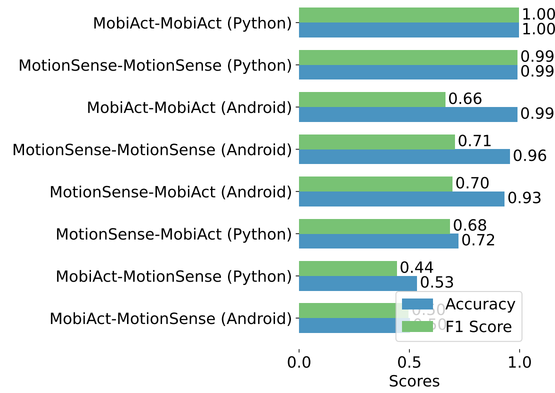

Figure 5: Performance of 4500 samples configuration with no pretrained layers before FL-TL

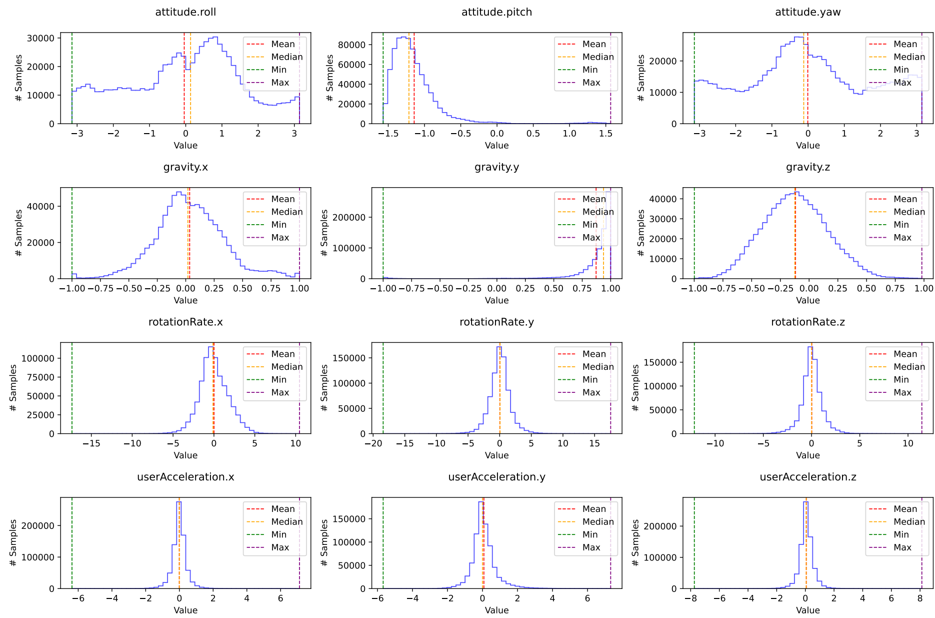

Figure 6: Non normalized attributes distribution of MotionSense dataset

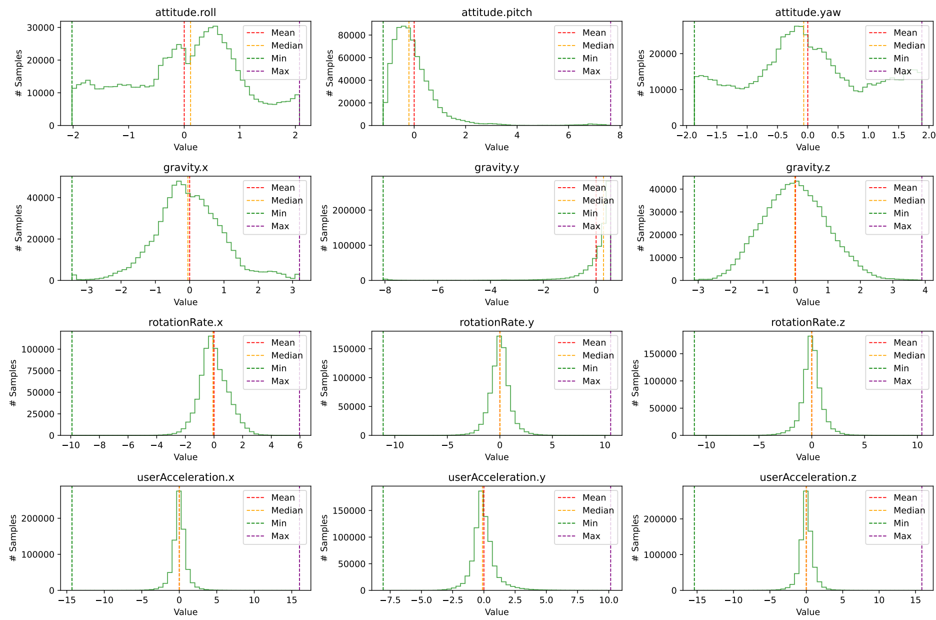

Figure 7: Normalized attributes distribution of MotionSense dataset

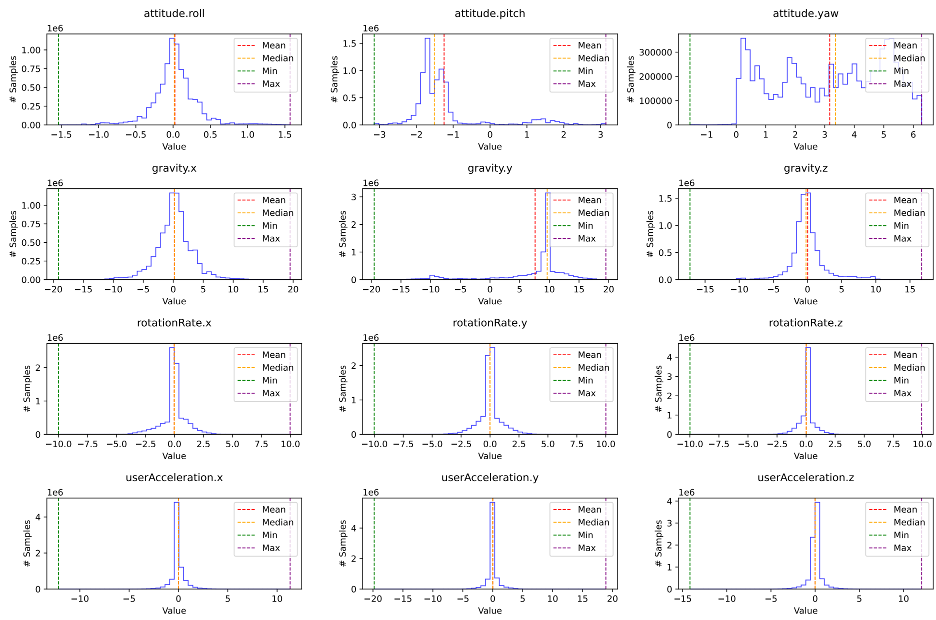

Figure 8: Non normalized attributes distribution of MobiAct dataset

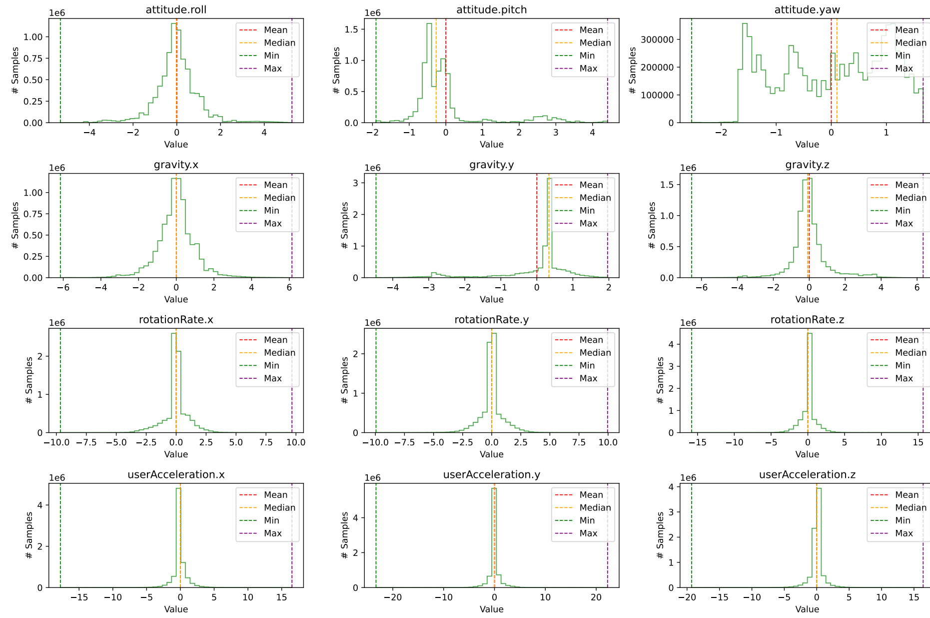

Figure 9: Normalized attributes distribution of MobiAct dataset

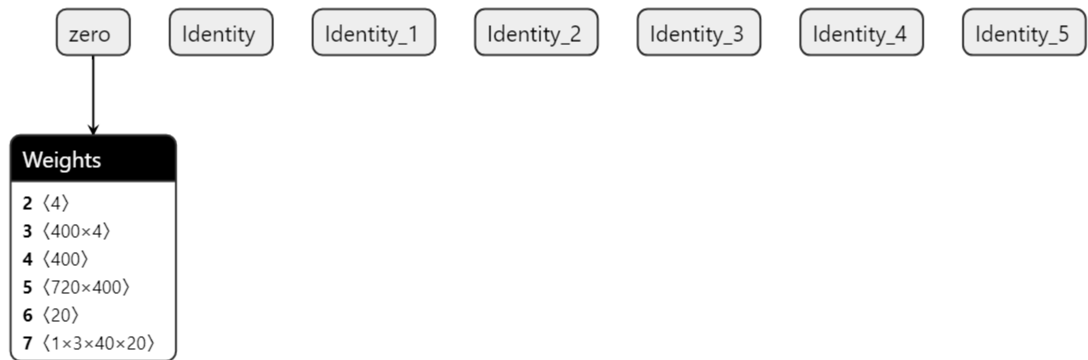

Figure 10: The initialization file setting up the initial parameters.

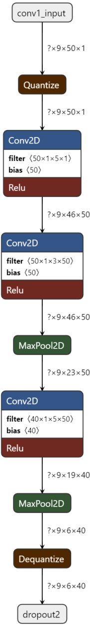

Figure 11: The bottleneck model.

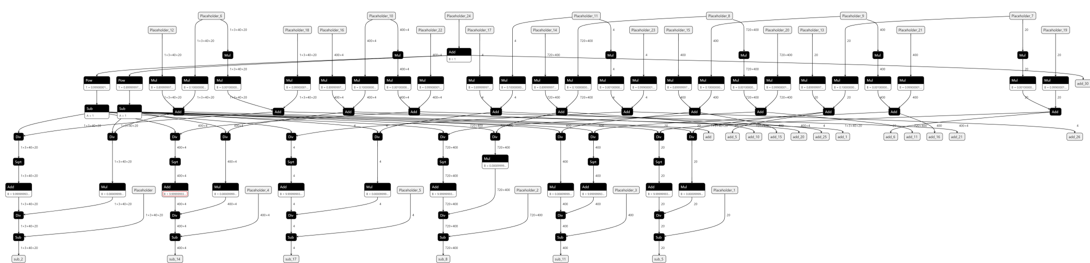

Figure 12: The optimizer file utilized for optimization

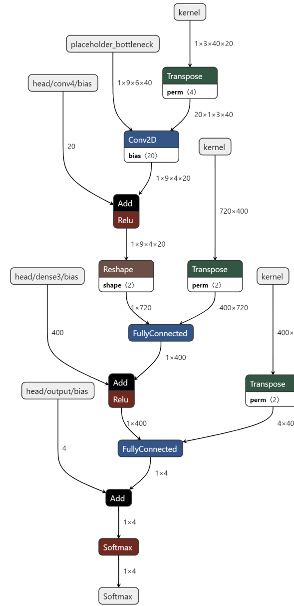

Figure 13: The inference file generating predictions.

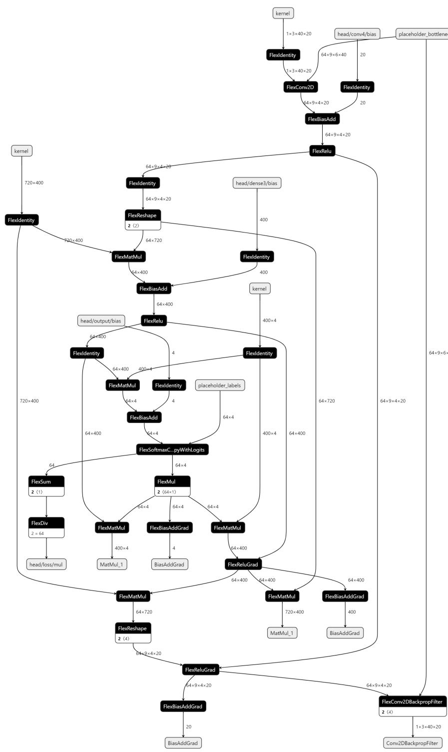

Figure 14: The training head model

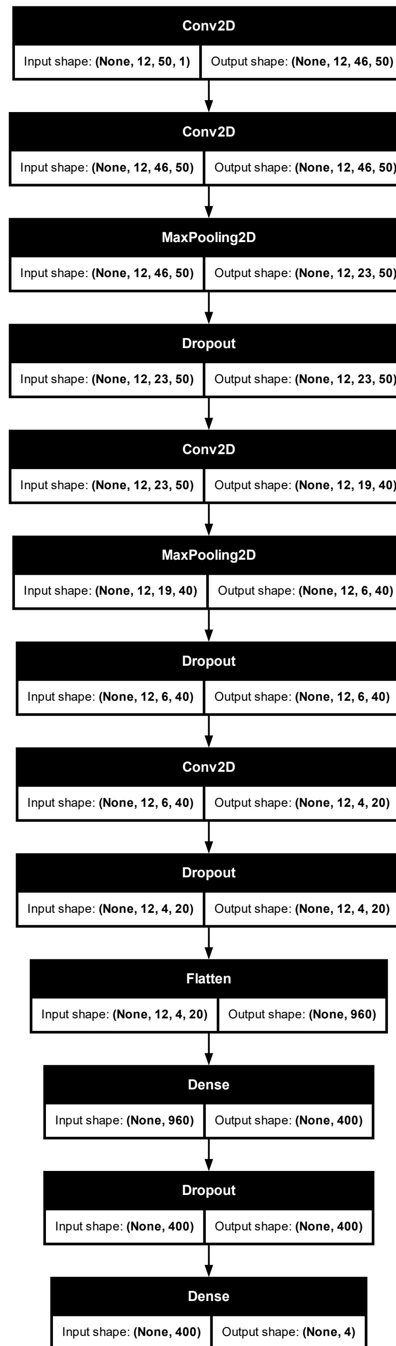

Figure 15: A customized version of the model proposed in [51].

Bibliography

[1] T Saponas, Jonathan Lester, Jon Froehlich, James Fogarty, and James Landay. «ilearn on the iphone: Real-time human activity classification on commodity mobile phones». In: *University of Washington CSE Tech Report UW-CSE-08-04-02* 2008 (2008) (cit. on pp. 1, 19).

[2] Jesús Fontecha, Fco Navarro, Ramón Hervás, and José Bravo. «Elderly Frailty Detection by using Accelerometer-Enabled Smartphones and Clinical Information Records». In: *Personal and Ubiquitous Computing* 17 (May 2012). DOI: 10.1007/s00779-012-0559-5 (cit. on p. 1).

[3] Jiangpeng Dai, Xiaole Bai, Zhimin Yang, Zhaohui Shen, and Dong Xuan. «PerFallD: A Pervasive Fall Detection System Using Mobile Phones». In: Oct. 2010, pp. 292–297. DOI: 10.1109/PERCOMW.2010.5470652 (cit. on p. 1).

[4] Wei Niu, Jiao Long, D. Han, and Yuanfang Wang. «Human activity detection and recognition for video surveillance». In: July 2004, 719–722 Vol.1. DOI: 10.1109/ICME.2004.1394293 (cit. on pp. 1, 3).

[5] Sharare Zehtabian, Siavash Khodadadeh, Ladislau Bölöni, and Damla Turgut. «Privacy-preserving learning of human activity predictors in smart environments». In: *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE. 2021, pp. 1–10 (cit. on pp. 1, 2, 21).

[6] White House. «Consumer Data Privacy in a Networked World: A Framework for Protecting A Privacy and Promoting Innovation in the GlobaEconom». In: *http://www. whitphi) nse pnY/siles/default/files/privac* (2012) (cit. on pp. 1, 20).

[7] Sumit Majumder and M.J. Deen. «Smartphone Sensors for Health Monitoring and Diagnosis». In: *Sensors* 19 (May 2019), p. 2164. DOI: 10.3390/s19092164 (cit. on pp. 1, 20).

[8] 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*. PMLR. 2017, pp. 1273–1282 (cit. on pp. 2, 20).

[9] Oscar D Lara and Miguel A Labrador. «A survey on human activity recognition using wearable sensors». In: *IEEE communications surveys & tutorials* 15.3 (2012), pp. 1192–1209 (cit. on pp. 4, 19).

[10] GSMA. *The Mobile Economy 2020*. https://www.gsma.com/solutions-and-impact/connectivity-for-good/mobile-economy/wp-content/uploads/2020/03/GSMA_MobileEconomy2020_Global.pdf (cit. on p. 5).

[11] MEMS Exchange. *What is MEMS?* https://www.mems-exchange.org/MEMS/what-is.html (cit. on p. 6).

[12] Salvador García, Julián Luengo, Francisco Herrera, et al. *Data preprocessing in data mining*. Vol. 72. Springer, 2015 (cit. on pp. 6, 19).

[13] Muhammad Shoaib, Hans Scholten, and Paul JM Havinga. «Towards physical activity recognition using smartphone sensors». In: *2013 IEEE 10th international conference on ubiquitous intelligence and computing and 2013 IEEE 10th international conference on autonomic and trusted computing*. IEEE. 2013, pp. 80–87 (cit. on p. 8).

[14] Muhammad Shoaib, Stephan Bosch, Ozlem Durmaz Incel, Hans Scholten, and Paul JM Havinga. «Fusion of smartphone motion sensors for physical activity recognition». In: *Sensors* 14.6 (2014), pp. 10146–10176 (cit. on p. 8).

[15] Arturo Moncada-Torres, Kaspar Leuenberger, Roman Gonzenbach, Andreas Luft, and Roger Gassert. «Activity classification based on inertial and barometric pressure sensors at different anatomical locations». In: *Physiological measurement* 35.7 (2014), p. 1245 (cit. on p. 8).

[16] Amazon Web Services. *What is a Neural Network?* https://aws.amazon.com/it/what-is/neural-network/ (cit. on p. 9).

[17] IBM. *Convolutional Neural Networks*. https://www.ibm.com/topics/convolutional-neural-networks (cit. on p. 10).

[18] Daniel J Beutel et al. «Flower: A friendly federated learning research framework». In: *arXiv preprint arXiv:2007.14390* (2020) (cit. on pp. 14, 21).

[19] NVIDIA. *TensorFlow*. https://www.nvidia.com/en-us/glossary/tensorflow/ (cit. on p. 16).

[20] TensorFlow. *Post-Training Quantization*. https://www.tensorflow.org/lite/performance/post_training_quantization (cit. on p. 17).

[21] Isabelle Guyon and André Elisseeff. «An introduction to variable and feature selection». In: *Journal of Machine Learning Research* 3.Mar (2003), pp. 1157–1182 (cit. on pp. 19, 20).

[22] Seungeun Chung, Jiyoun Lim, Kyoung Ju Noh, Gague Kim, and Hyuntae Jeong. «Sensor data acquisition and multimodal sensor fusion for human activity recognition using deep learning». In: *Sensors* 19.7 (2019), p. 1716 (cit. on p. 19).

[23] Oresti Banos, Juan-Manuel Galvez, Miguel Damas, Hector Pomares, and Ignacio Rojas. «Window size impact in human activity recognition». In: *Sensors* 14.4 (2014), pp. 6474–6499 (cit. on p. 19).

[24] Muhammad Shoaib, Stephan Bosch, Ozlem Durmaz Incel, Hans Scholten, and Paul JM Havinga. «Complex human activity recognition using smartphone and wrist-worn motion sensors». In: *Sensors* 16.4 (2016) (cit. on pp. 19, 40).

[25] Wesllen Sousa, Eduardo Souto, Jonatas Rodrigres, Pedro Sadarc, Roozbeh Jalali, and Khalil El-Khatib. «A comparative analysis of the impact of features on human activity recognition with smartphone sensors». In: *Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web*. 2017, pp. 397–404 (cit. on pp. 19, 20, 39).

[26] Wesllen Sousa Lima, Eduardo Souto, Khalil El-Khatib, Roozbeh Jalali, and Joao Gama. «Human activity recognition using inertial sensors in a smartphone: An overview». In: *Sensors* 19.14 (2019), p. 3213 (cit. on p. 19).

[27] Uwe Maurer, Asim Smailagic, Daniel P Siewiorek, and Michael Deisher. «Activity recognition and monitoring using multiple sensors on different body positions». In: *International Workshop on Wearable and Implantable Body Sensor Networks (BSN'06)*. IEEE. 2006, 4–pp (cit. on pp. 19, 40).

[28] Kun Xia, Jianguang Huang, and Hanyu Wang. «LSTM-CNN architecture for human activity recognition». In: *IEEE Access* 8 (2020), pp. 56855–56866 (cit. on pp. 19, 20).

[29] Nidhi Dua, Shiva Nand Singh, and Vijay Bhaskar Semwal. «Multi-input CNN-GRU based human activity recognition using wearable sensors». In: *Computing* 103.7 (2021), pp. 1461–1478 (cit. on pp. 19, 20).

[30] Ian Goodfellow. *Deep learning*. 2016 (cit. on p. 20).

[31] Friedrich Foerster, Manfred Smeja, and Jochen Fahrenberg. «Detection of posture and motion by accelerometry: a validation study in ambulatory monitoring». In: *Computers in human behavior* 15.5 (1999), pp. 571–583 (cit. on p. 20).

[32] Harikrishna Narasimhan, Weiwei Pan, Purushottam Kar, Pavlos Protopapas, and Harish G Ramaswamy. «Optimizing the multiclass F-measure via biconcave programming». In: *2016 IEEE 16th international conference on data mining (ICDM)*. IEEE. 2016, pp. 1101–1106 (cit. on pp. 20, 40).

[33] Rinat Khusainov, Djamel Azzi, Ifeyinwa E Achumba, and Sebastian D Bersch. «Real-time human ambulation, activity, and physiological monitoring: Taxonomy of issues, techniques, applications, challenges and limitations». In: *Sensors* 13.10 (2013), pp. 12852–12902 (cit. on p. 20).

[34] E.T. McAdams et al. «Biomedical Sensors for Ambient Assisted Living». In: *Advances in Biomedical Sensing, Measurements, Instrumentation and Systems*. Berlin, Germany: Springer, 2010, pp. 240–262 (cit. on p. 20).

[35] Lei Gao, AK Bourke, and John Nelson. «Evaluation of accelerometer based multi-sensor versus single-sensor activity recognition systems». In: *Medical engineering & physics* 36.6 (2014), pp. 779–785 (cit. on p. 20).

[36] Shibo Zhang, Yaxuan Li, Shen Zhang, Farzad Shahabi, Stephen Xia, Yu Deng, and Nabil Alshurafa. «Deep learning in human activity recognition with wearable sensors: A review on advances». In: *Sensors* 22.4 (2022), p. 1476 (cit. on p. 20).

[37] Chang Wang and Sridhar Mahadevan. «Heterogeneous domain adaptation using manifold alignment». In: *IJCAI proceedings-international joint conference on artificial intelligence*. Vol. 22. 1. Citeseer. 2011, p. 1541 (cit. on p. 20).

[38] Lixin Duan, Dong Xu, and Ivor Tsang. «Learning with augmented features for heterogeneous domain adaptation». In: *arXiv preprint arXiv:1206.4660* (2012) (cit. on p. 20).

[39] Maayan Harel and Shie Mannor. «Learning from multiple outlooks». In: *arXiv preprint arXiv:1005.0027* (2010) (cit. on p. 20).

[40] Dong Yang et al. «Federated semi-supervised learning for COVID region segmentation in chest CT using multi-national data from China, Italy, Japan». In: *Medical image analysis* 70 (2021), p. 101992 (cit. on p. 20).

[41] 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». In: *arXiv preprint arXiv:1811.03604* (2018) (cit. on p. 20).

[42] Farzaneh Shoeleh and Masoud Asadpour. «Graph based skill acquisition and transfer learning for continuous reinforcement learning domains». In: *Pattern Recognition Letters* 87 (2017), pp. 104–116 (cit. on p. 21).

[43] Mohammed Elhenawy, Huthaifa I Ashqar, Mahmoud Masoud, Mohammed H Almannaa, Andry Rakotonirainy, and Hesham A Rakha. «Deep transfer learning for vulnerable road users detection using smartphone sensors data». In: *Remote Sensing* 12.21 (2020), p. 3508 (cit. on p. 21).

[44] Jian Cheng, Jiaxiang Wu, Cong Leng, Yuhang Wang, and Qinghao Hu. «Quantized CNN: A unified approach to accelerate and compress convolutional networks». In: *IEEE transactions on neural networks and learning systems* 29.10 (2017), pp. 4730–4743 (cit. on p. 21).

[45] Ioan Lucan Orăşan, Ciprian Seiculescu, and Cătălin Daniel Caleanu. «Benchmarking tensorflow lite quantization algorithms for deep neural networks». In: *2022 IEEE 16th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE. 2022, pp. 000221–000226 (cit. on pp. 21, 48, 49).

[46] Yiqiang Chen, Xin Qin, Jindong Wang, Chaohui Yu, and Wen Gao. «Fedhealth: A federated transfer learning framework for wearable healthcare». In: *IEEE Intelligent Systems* 35.4 (2020), pp. 83–93 (cit. on p. 22).

[47] I Kevin, Kai Wang, Xiaokang Zhou, Wei Liang, Zheng Yan, and Jinhua She. «Federated transfer learning based cross-domain prediction for smart manufacturing». In: *IEEE Transactions on Industrial Informatics* 18.6 (2021), pp. 4088–4096 (cit. on p. 22).

[48] Flower. *How to Implement Strategies*. `https://flower.ai/docs/framework/how-to-implement-strategies.html` (cit. on p. 36).

[49] Android Developers. *Sensors Overview*. `https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview` (cit. on p. 39).

[50] API Levels. *API Levels for Android Versions*. `https://apilevels.com/` (cit. on p. 39).

[51] Mohammad Malekzadeh, Richard G. Clegg, Andrea Cavallaro, and Hamed Haddadi. «Protecting Sensory Data Against Sensitive Inferences». In: *Proceedings of the 1st Workshop on Privacy by Design in Distributed Systems*. W-P2DS'18. Porto, Portugal: ACM, 2018, 2:1–2:6. ISBN: 978-1-4503-5654-1. DOI: `10.1145/3195258.3195260`. URL: `http://doi.acm.org/10.1145/3195258.3195260` (cit. on pp. 45, 68).

[52] Frirhos. *PHAR Repository*. Accessed: 2024-09-24. 2024. URL: `https://github.com/Frirhos-he/PHAR` (cit. on pp. 46, 52).