UNIVERSITY OF FLORENCE,
IMT SCHOOL FOR ADVANCED STUDIES LUCCA

MD2SL
Second Level Master Course in
Data Science and Statistical Learning

FEDERATED HUMAN ACTIVITY RECOGNITION

DARIO COMANDUCCI

Supervisor: *Prof. Fabio Pinelli*

Academic year 2023-2024

CONTENTS

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Prof. Fabio Pinelli, for his guidance and input throughout this journey. I also extend my thanks to my friends, not only for their understanding during my prolonged absence throughout the MD2SL study period, but also for their huge patience and support in many other situations.

A special dedication goes to my family for their strength and resilience in the difficult period they have been enduring.

Thank you all,
D. C.

ABSTRACT

This research explores Human Activity Recognition (HAR) through smartphone-based sensors, focusing on accelerometers and gyroscopes, now integral to modern devices.

The work is structured into three main areas:

1. A review of key HAR approaches, emphasizing supervised learning with recurrent networks (RNNs, LSTMs) and MLP (Multi-Layer Perceptron) classifiers;

2. Extending these methods to a federated learning framework, where models are trained locally on device-specific time series and then aggregated into a global model;

3. Identifying suitable datasets to evaluate the proposed techniques.

A combination of a recurrent model with a MLP classifier was adopted here to address the HAR task. The recurrent model's internal state produces embedding vectors encoding signal information for performed activities. These embeddings, processed by the MLP, could also be utilized in clustering tasks.

Federated learning tackles critical HAR challenges more effectively than a centralized approach. A significant issue is the need for diverse user data collected across multiple devices, which imposes storage constraints on centralized systems managing large-scale datasets. Moreover, smartphone-generated user data is highly susceptible to privacy concerns. Federated learning mitigates this problem by ensuring that all data remains securely on the user's device, reducing privacy risks while promoting decentralization. The federated learning implementation utilized the Flower library [1].

A benchmark dataset [13][14], unifying six datasets from five distinct institutions, was used to evaluate these techniques. The unified datasets share at least four core activities across six categories (e.g., sitting, standing, walking, running, go up/down the stairs) with standardized sampling rates and units. Experimental results confirm the effectiveness of federated learning in HAR applications.

A comprehensive overview of the research conducted can be summarized as follows. Chapter 1 reviews HAR literature and tools, establishing the foundation for the study. Chapter 2 explains the classification model and its integration. Chapter 3 introduces federated learning and the Flower library, while Chapter 4 presents results from both centralized and federated settings. Finally, Chapter 5 discusses findings and future directions.

1 INTRODUCTION

DIGITAL PHENOTYPING is a multidisciplinary field focused on quantifying the *human phenotype* in real-time at the individual level. Using data from personal digital devices, it aims to deepen our understanding of human behavior and health. In this context, the *human phenotype* refers to observable and measurable traits reflecting a person's behavior and activities. These traits include:

- *Spatial position* (identifying where an individual is at a given moment) and *inertial measurements* (capturing dynamics like acceleration and angular momentum);
- *Physical movements* (e.g., walking, running, sitting, etc.);
- *Specific activities* (e.g., climbing stairs, driving, exercising, etc.).

Within the broader scope of digital phenotyping, the research focus of interest here is identified as *Human Activity Recognition*.

1.1 HUMAN ACTIVITY RECOGNITION

Human Activity Recognition (HAR) is among the most promising research topics across various fields, including pervasive and mobile computing, surveillance-based security, context-aware computing, and ambient assistive living. It is a complex process, characterized by four fundamental aspects [3]:

- Selecting and implementing appropriate sensors on objects and environments to monitor and capture a user's behavior along with changes in the state of the environment;
- collecting, storing, and processing the perceived information through data analysis techniques and/or knowledge representation formalisms at appropriate levels of abstraction;
- Creating computational activity models to enable systems or software agents to perform reasoning and manipulations;
- Selecting or developing reasoning algorithms to infer activities from sensor data.

1.1.1 *Sensors for HAR*

There are mainly two types of HAR: video-based HAR and sensor-based HAR. Video-based HAR analyzes videos or images containing

human movements captured by cameras, while sensor-based HAR focuses on motion data collected from smart sensors such as accelerometers, gyroscopes, Bluetooth, sound sensors, and so on [27]. With the rapid advancements in sensor technology and pervasive computing, sensor-based HAR has seen widespread adoption and growing popularity. This approach, which also emphasizes robust privacy protection, will be explored further in the following discussion.

The idea of using sensors for activity monitoring and recognition has existed since the late 1990s. Initially, it was introduced and experimented with-in the context of home automation and various location-based applications aimed at adapting systems to users' positions. The approach was soon deemed more useful and suitable in the field of ubiquitous and mobile computing (an emerging area in the late 1990s as well) due to its ease of implementation [3].

Most of the research during that period focused on wearable sensors, including dedicated sensors attached to the human body and portable devices such as mobile phones, with applications in ubiquitous computing scenarios like providing context-aware mobile devices. The activities monitored in these studies were primarily physical activities such as movement, walking, and running [3].[1]

Wearable Sensors

Wearable sensors typically refer to devices placed directly or indirectly on the human body. They generate signals as the user performs activities, enabling the monitoring of physiological states or movement traits. These sensors can be embedded in clothing, glasses, belts, shoes, wristwatches, mobile devices, or positioned directly on the body. Different types of sensory data prove effective in classifying various activities [3].

ACCELEROMETERS These sensors, often combined with gyroscopes and magnetometers [27], are widely used in wearable activity monitoring. By detecting changes in acceleration and angular velocity caused by body movements, they effectively infer activities, especially repetitive ones like walking, running, sitting, standing, and climbing stairs.

GPS Global Positioning System (GPS) sensors are another widely used type of wearable device for monitoring location-based activities in outdoor pervasive and mobile environments.

1 In the early 2000s another sensor-based approach emerged, utilizing sensors applied to objects to monitor human activities. Known as the "dense sensing" approach, it recognizes activities by inferring user-object interactions and is well-suited for tasks involving multiple objects or instrumental activities of daily living [3][27]. Unlike wearable sensor-based methods, dense sensing dominates smart environment applications, such as Ambient Assisted Living (AAL), particularly for monitoring, recognizing, and assisting activities to support independent and active living for elderly users within their homes.

BIOSENSORS This type of sensors is designed to monitor activities through vital signs. Various forms of sensors have been studied to measure a wide range of vital signs, including blood pressure, heart rate, EEG, ECG, and respiratory information.

Wearable sensor-based activity monitoring faces practical challenges, such as the need for hands-free, continuously operational devices and issues related to battery life, size, ease of use, and user adoption; smartphones, widely integrated into daily life, offer a promising alternative: With their affordability and ongoing technological advancements, they offer effective tools for activity monitoring and recognition [3].

Anyway, wearable sensors are not ideal for monitoring activities with complex physical motions or multiple environmental interactions. Sometimes, their data alone isn't enough to differentiate even simple physical activities (e.g., making tea or making coffee).

1.1.2 *Approaches to HAR problems*

Pattern Recognition Approaches

HAR can be addressed as a typical pattern recognition problem using machine learning algorithms such as decision trees, Support Vector Machines, naïve Bayes and Hidden Markov Models; however, conventional pattern recognition methods have notable drawbacks [27].

First, feature extraction is often heuristic and manual, heavily relying on human expertise or domain knowledge. While useful in specific contexts, this approach is less effective and more time-consuming for general environments and tasks.

Second, conventional methods can only capture shallow features, like statistical data (e.g., mean, variance, frequency, amplitude). These are sufficient for recognizing low-level activities (e.g., walking, running) but struggle with high-level or context-aware activities (e.g., drinking coffee), which are more complex.

Third, pattern recognition methods typically require large volumes of labeled data for model training. Since most activity data in real-world applications remains unlabeled, these models perform poorly in unsupervised learning tasks. In contrast, deep generative networks can leverage unlabeled data for training.

Deep Learning Approaches

Deep learning techniques effectively address these limitations: Methods such as deep neural networks, convolutional neural networks, autoencoders, and recurrent neural networks minimize the need for manual feature design by enabling higher-level feature learning through end-to-end neural network architectures [27]. Feature extraction and model construction are often performed simultaneously in deep learning models, with features being automatically learned through the

network rather than manually designed. Additionally, deep neural networks can extract high-level representations in deeper layers, making them better suited for complex activity recognition tasks.

For simple physical activities and data from inertial sensors in wearable devices, such as accelerometers and gyroscopes, supervised learning approaches are the most effective. However, these methods require large amounts of labeled data for model training, posing challenges due to the cost, invasiveness, and time-consuming nature of data collection and annotation [17]: to overcome these issues, alternative approaches using limited labeled data have been proposed for accurate HAR, such as *self-supervised* learning algorithms.

1.1.3 *A Formal Definition for Supervised HAR*

Formally, let us assume that a user is performing activities belonging to a predefined set $\mathcal{A} = \{A_i\}_{i=1}^{m}$, where A_i represents an activity category, and m is the total number of categories considered [27].

Let $\mathbf{s} = \{\mathbf{d}_1 \dots \mathbf{d}_t \dots \mathbf{d}_n\}$ be a sequence of sensor readings capturing information about the performed activities, where $\mathbf{d}_t$ denotes the sensor reading at time t (with $n \geqslant m$).

The goal is to build a model $F(\mathbf{s})$ capable of predicting the sequence $\hat{A} = \{\hat{A}_j\}_{j=1}^{n}$ (with $\hat{A}_j \in \mathcal{A}$):

$$\hat{A} = \{\hat{A}_j\}_{j=1}^{n} = F(\mathbf{s})$$

To learn $F(\mathbf{s})$, the discrepancy between the predicted activities in $\hat{A}$ and the actual activities in A is minimized. Typically, a positive loss function $L(\mathbf{s}, A)$ is constructed to reflect this discrepancy.

Usually, $F(\cdot)$ does not directly take $\mathbf{s}$ as input; it is often assumed that a projection function $\Phi(\cdot)$ maps the sensor readings $\mathbf{d}_i \in \mathbf{s}$ into a d-dimensional feature vector $\Phi(\mathbf{d}_i) \in \mathbb{R}^d$. Consequently, the objective transforms into minimizing the loss function $L\big(\{\Phi(\mathbf{d}_j)\}_{j=1}^{n}, A\big)$.

1.2 DATASETS FOR HAR

Several datasets have been created in the literature to enable the evaluation of HAR models, generally used in isolation. Some datasets are extensive and encompass a wide range of activities collected in real-world scenarios, while others are smaller and include a limited number of activities in controlled environments. These datasets include various users, activities, and sensors, providing a good starting point for HAR model development [13].

Although these datasets present good variability, it is important to note that each dataset constitutes a domain due to *bias* introduced by data collection protocols, demographic settings, and other factors. Therefore, it is essential to evaluate model generalization capabilities across different datasets to ensure they perform well in new

scenarios—this is a very challenging task partly due to the lack of a standardized approach for assessing model generalization across different datasets, as they have different units of measurement, sampling frequencies, as well as activities and labels that are not shared among them, making it difficult to evaluate model generalization capabilities across datasets [13].

1.2.1 *The DAGHAR Benchmark*

Recently, the DAGHAR benchmark [14, 13] was proposed, which gathers various public datasets containing raw inertial sensor data (particularly tri-axial accelerometers and tri-axial gyroscopes) exclusively from smartphones, making them compliant with a common standard regarding accelerometer measurement units, sampling frequency, gravitational component[2], activity labels, user partitioning, and time window size.[3]

The study in [13] conducted an in-depth investigation of over 40 HAR datasets, gathering key information such as the number of samples, types of recorded activities, participant demographics, sampling frequencies, sensor characteristics, citation frequency, and data collection protocols. From this initial list, datasets were selected based on the following criteria:

- Availability of raw data, not just preprocessed data, to ensure flexibility in analysis and preprocessing;
- Data integrity, meaning the dataset must[4]
  - include timestamps,
  - be free of missing values, inconsistencies, or irregularities,
  - have at least one associated research publication,
  - originate from smartphones,
  - include both accelerometer and gyroscope data;
- Inclusion of a substantial set of shared activities, ensuring the dataset supports a wide range of activity recognition tasks;
- Focus on normal daily human activities, excluding datasets that primarily feature sports, geography-based activities, or require invasive data collection methods, such as microphone usage.

2 The constant Earth gravity (approximately 9.81 m/s^2) is recorded by accelerometers, but it must be separated from actual motion data to correctly analyze human activities: separating the gravitational component from accelerometer data can improve the accuracy and consistency of physical activity measurements, allowing better generalization of HAR models.

3 Other datasets are listed in appendix **??**

4 These sub-criteria were deemed essential to ensure data reliability and suitability for further analysis.

Regarding the last two points, datasets were selected that contain at least four activities among "Sit" (sitting), "Stand" (standing), "Walk" (walking), "Upstairs" (ascending stairs), "Downstairs" (descending stairs), and "Run" (running); the datasets included in the benchmark report at least four of these activities (Tab. 1.1):[5]

- KuHAR (version 5, KH) [12]
- MotionSense (MS) [8, 9]
- RealWorld (RW) [15, 22]
- WISDM (WIreless Sensor Data Mining) dataset (WDM) [7, 6, 19]
- UCI HAR dataset (UCI) [20, 28]

Signal Characteristics

SAMPLING FREQUENCY Regarding the sampling frequency, an observation from [13] about the WISDM dataset is worth mentioning: a trimodal distribution at 50 Hz, 25 Hz, and 20 Hz, with less than half of the samples corresponding to the nominal sampling frequency (20 Hz) was noted; this discrepancy could be due to differences in the devices used during data recording (WISDM utilized three different smartphone models).[6] Additionally, "due to the nature of the Android operating system, the sampling frequency is only taken as a suggestion, so actual sampling frequencies sometimes differ" [28].

ACCELEROMETERS Another relevant aspect to consider is whether the accelerometer signal (Acc) includes gravitational acceleration. Acc reflects total acceleration, combining body acceleration (sensor movement) and gravity (affecting all axes during rotation). To separate the two, procedures like a low-order high-pass Butterworth filter (e.g., order 3, cutoff <1 Hz) are typically applied. Furthermore, similar to previous observations regarding sampling frequency, the datasets again differ significantly:

- For datasets with only body acceleration (KuHar and UCI), data were processed without the gravitational component.
- For MotionSense, which provides body and gravity acceleration separately, both signals were summed.
- For datasets with total acceleration, this signal was used as is.

5 Another noteworthy dataset is ExtraSensory [24, 25], probably the largest publicly available dataset. One of its key features is the "in-the-wild" data collection protocol: data was collected from users engaged in their normal natural behavior, leaving the labeling task to the users themselves. This choice led to a significant imbalance in the number of samples per activity and lower confidence in data labels [13].

6 In [13], data were interpolated with splines to match the nominal frequency.

Table 1.1. List of codes: “KH”, KuHar dataset; “MS”, MotionSense; “RW”, RealWorld; “WDM”, WISDM dataset; “UCI”: UCI-HAR. The RealWorld dataset uses multiple smartphones positioned in different locations during data collection (e.g., thigh, waist, shin, head, etc.): Only data collected from sensors placed on the thigh (RW-Thigh) and waist (RW-Waist) were considered, which should be equivalent to the pocket and fanny pack positions.

		KH	MS	RW	UCI	WDM
Activities	Sit (code 0)	•	•	•	•	•
	Stand (code 1)	•	•	•	•	•
	Walk (code 2)	•	•	•	•	•
	Upstairs (code 3)	•	•	•	•	
	Downstairs (code 4)	•	•	•	•	
	Run (code 5)	•	•	•		•
Sensors	smartphone position	pouch	pocket	thigh/waist	pocket	pouch
	gyroscope measurements	rad/s	rad/s	rad/s	rad/s	rad/s
	accelerometer measurements	m/s^2	G	m/s^2	m/s^2	G
	sampling frequency [Hz]	100	50	50	20	50
	# users (total)	79	24	15/15	30	51
	# users (train)	57	17	10/10	21	36
	# users (valid.)	7	2	2/2	3	4
	# users (test)	15	5	3/3	6	11

Signal Views

Once the data were selected, in [13] were created *views* of the dataset as a pair $(X, \mathbf{y})$, where X is an $N \times d$ matrix with N samples, each having a dimension d, and $\mathbf{y}$ is the corresponding set of labels. The original time series of each Acc/Gyr axis were divided into non-overlapping 3-second windows (the inertial sensor recordings in the datasets can range from a few seconds to several minutes).[7]

PRE-PROCESSING The adopted procedure involved processing the accelerometer (Acc) and gyroscope (Gyr) data:

- The data were converted to a sampling frequency of 20 Hz;
- The accelerometer unit was converted from G to m/s^2 for the MS and UCI datasets;
- The gravitational component was removed using a third-order high-pass Butterworth filter (0.3 Hz cutoff) on pre-processed data, as previously described for the accelerometer.[8]

Figs. 1.1–1.6 present a gallery of examples (one view per action); note the different scale order for "sit" and "stand" compared to others.

Data Partitioning

The adopted procedure involved processing the accelerometer (Acc) and gyroscope (Gyr) data, considering the sampling frequency of the various datasets and segmenting the raw time series into non-overlapping 3-second time windows.

Since both Acc and Gyr are tri-axial time series, the dimensionality of a baseline view sample is defined as $2 \cdot 3 \cdot (f_s \cdot 3)$, where f_s is the sampling frequency, and the x-y-z Acc time series are concatenated with the x-y-z Gyr time series.

The partitioning process was performed at the user level with a 70/10/20 ratio for the training, validation, and test sets, respectively. This means that for each dataset, users were partitioned into these three sets in such a way that all samples from a given user were contained within a single set, thereby preventing any mixing of samples

7 Although various studies have used fixed-duration windowing schemes ranging from as short as 0.5 seconds to more than 30 seconds, the most common interval for HAR activities is between 1 and 5 seconds. More specifically, a window duration between 2.5 and 3.5 seconds should allow a better balance between performance and latency for human activity recognition tasks [13].

8 For brevity, the description of an intermediate phase, the *baseline views*, was omitted. The processing in this phase was minimal (primarily the division into Training set, Validation set, and Test set with the same partitioning described in § 1.2.1) following a principle of "minimal interference." The purpose of the baseline views is to provide a benchmark to verify that the overall standardization procedure does not alter the informational content of the various datasets, comparing classifiers trained on baseline views versus standardized ones.

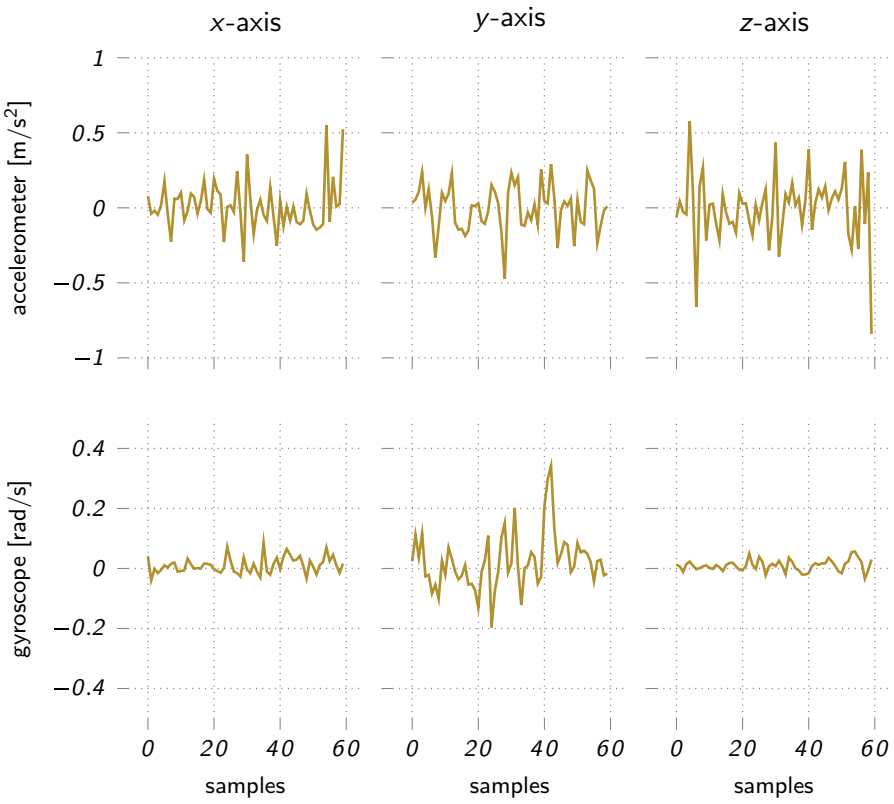


Figure 1.1. Example view for action "sit" (code 0).

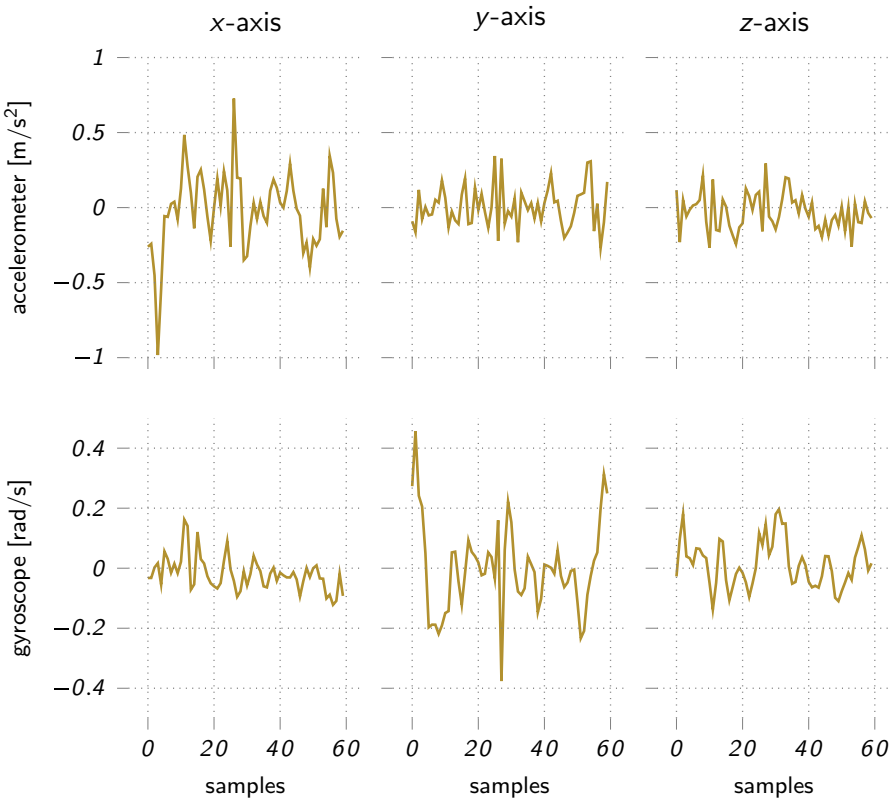


Figure 1.2. Example view for action "stand" (code 1).

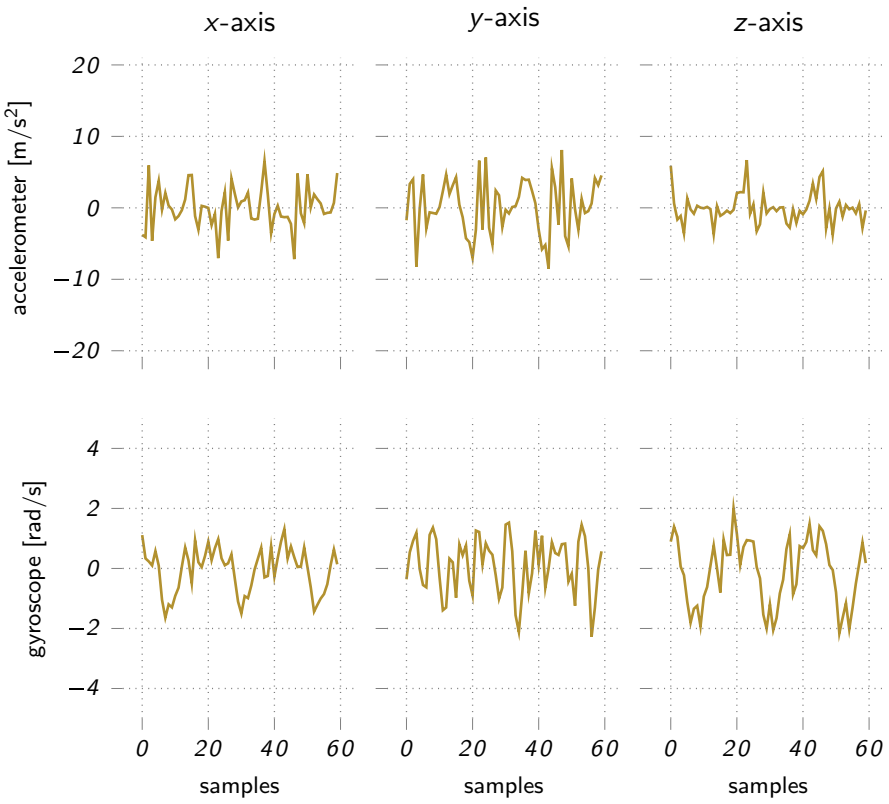


Figure 1.3. Example view for action "walk" (code 2).

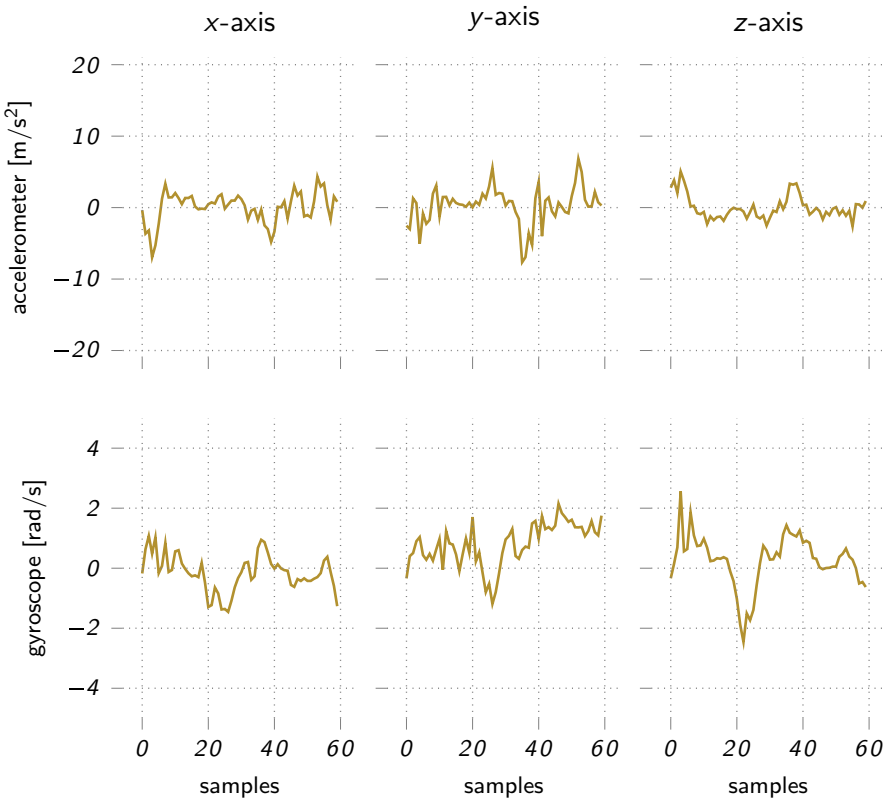


Figure 1.4. Example view for action "stairs up" (code 3).

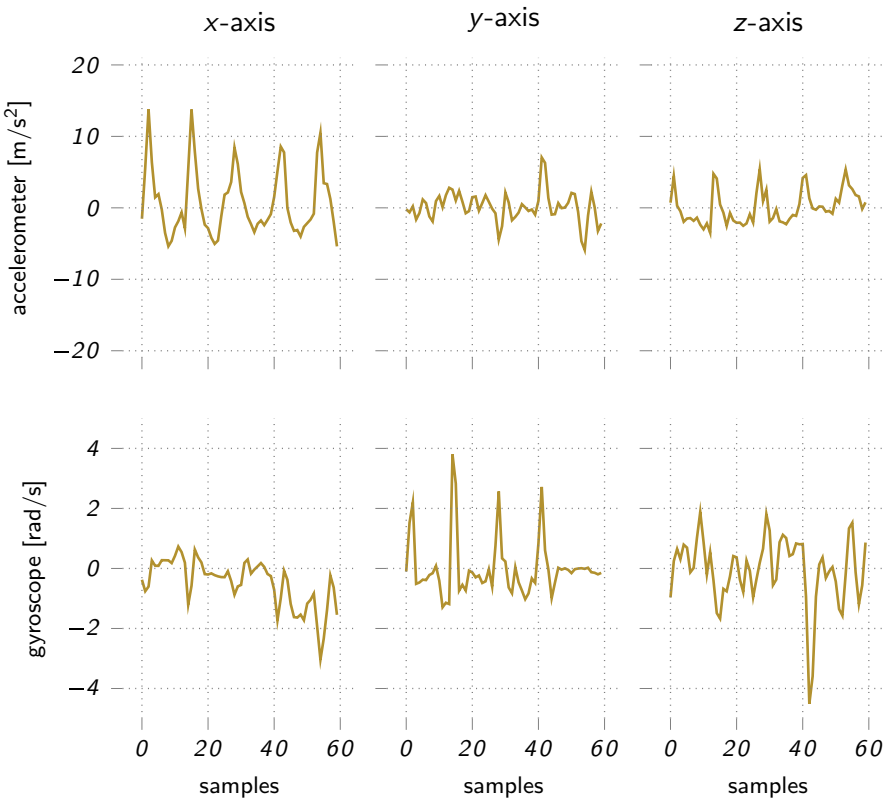


Figure 1.5. Example view for action "stairs down" (code 4).

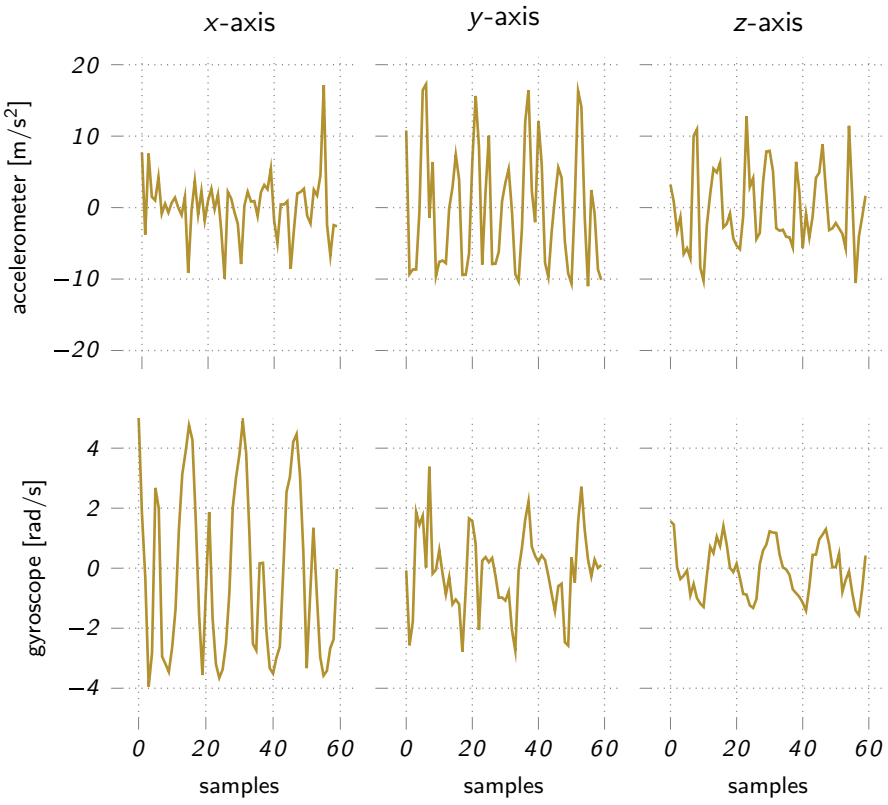


Figure 1.6. Example view for action "run" (code 5).

Table 1.2. List of codes: "KH", KuHar dataset; "MS", MotionSense; "RW", RealWorld; "WDM", WISDM dataset; "UCI": UCI-HAR. RW was later split into "RW-T" and "RW-W", separating acquisitions where the phone was positioned on the thigh (RW-Thigh) from those where the smartphone was worn at the waist (RW-Waist).

Set	Train	Validation	Test	Total
KH	1386 (70.9%)	144 (7.4%)	426 (21.8%)	1956
MS	3558 (70.6%)	420 (8.3%)	1062 (21.0%)	5040
RW-T	8400 (65.7%)	1764 (13.8%)	2628 (20.5%)	12792
RW-W	10332 (69.9%)	1854 (12.5%)	2592 (17.5%)	14778
UCI	2420 (70.1%)	340 (9.9%)	690 (20.0%)	3450
WDM	8736 (71.2%)	944 (7.7%)	2596 (21.1%)	12276

from the same user across different sets. However, since users may have different numbers of samples, the actual number of samples in each set may not strictly follow the 70/10/20 ratio.

After partitioning, the number of samples per activity within each set was balanced by randomly sampling the same number of samples for each activity, corresponding to the activity with the fewest samples in the set. Since the methodology was applied independently to each dataset, variations in the number of samples among different partitions may occur (Tab. 1.2).A detailed overview of the datasets is presented in Figs. 1.7–1.9, showing as heatmaps both the number and proportion of examples per user across the six action categories.[9]

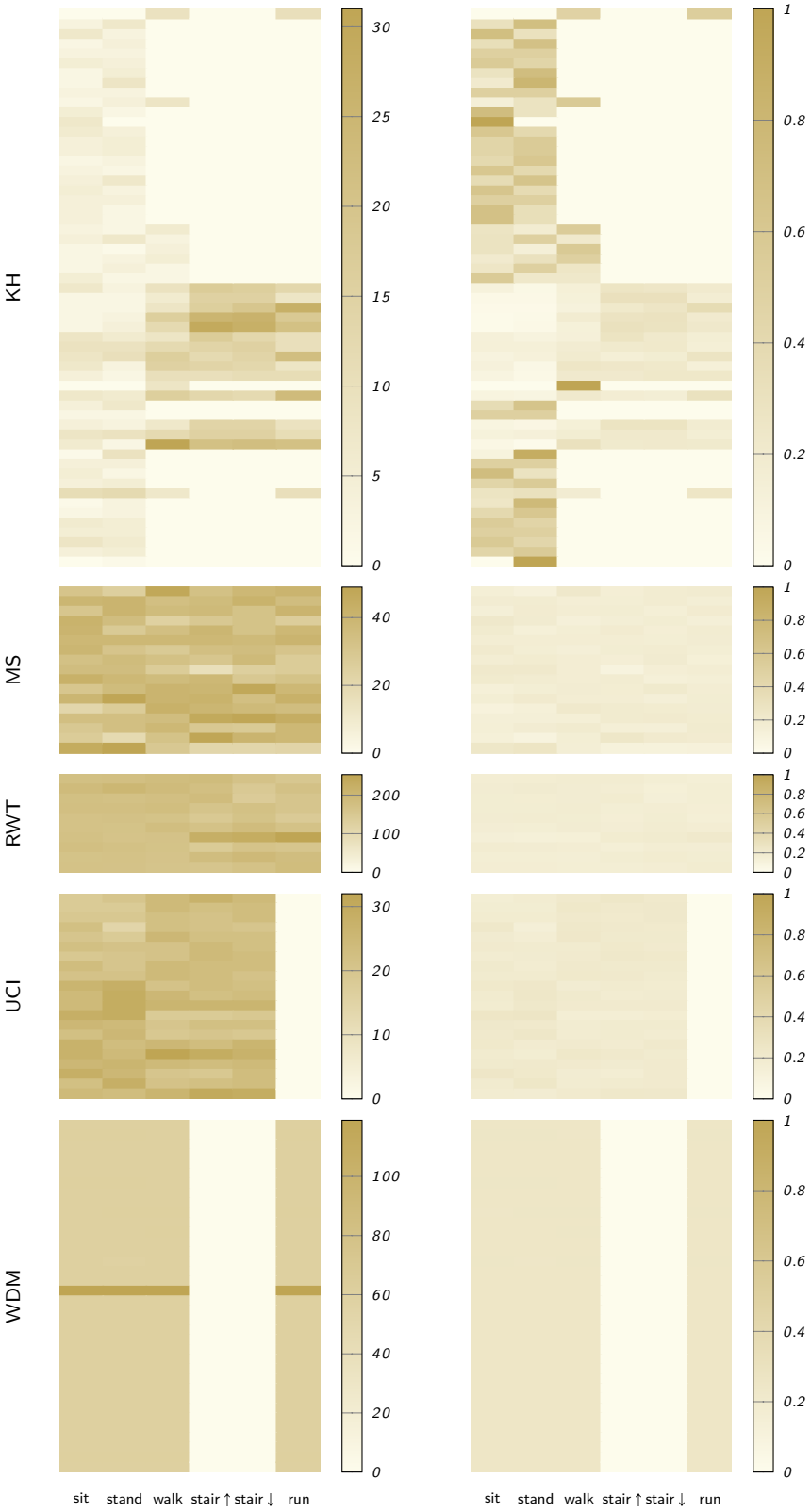
1.3 CRITICALITIES IN HAR

Considering the insights gained about HAR tasks, several challenges arise with a centralized approach:

- Firstly, to gather a substantial amount of examples, the ideal strategy involves collecting actions from a large number of users utilizing their personal devices, such as smartphones;
- In such cases, the vast amount of generated data creates storage challenges on a central server;
- Additionally, the inherently private nature of data collected via smartphones necessitates a less intrusive approach.

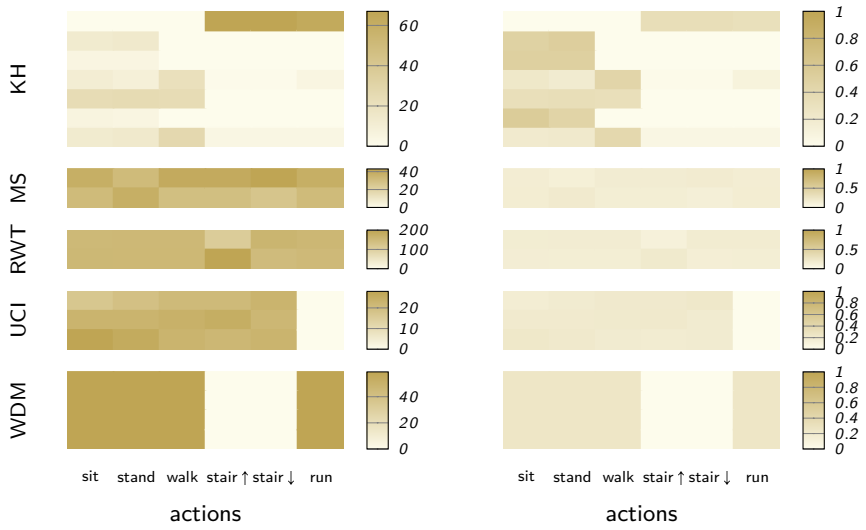
Federated learning (refer to Chapter 3) offers a promising solution to address these concerns and serves as the focal point of this study.

9 RW-W is omitted for space reasons; RW-T is reported instead since it will also be used later in the experiments (§ 4.2.2).



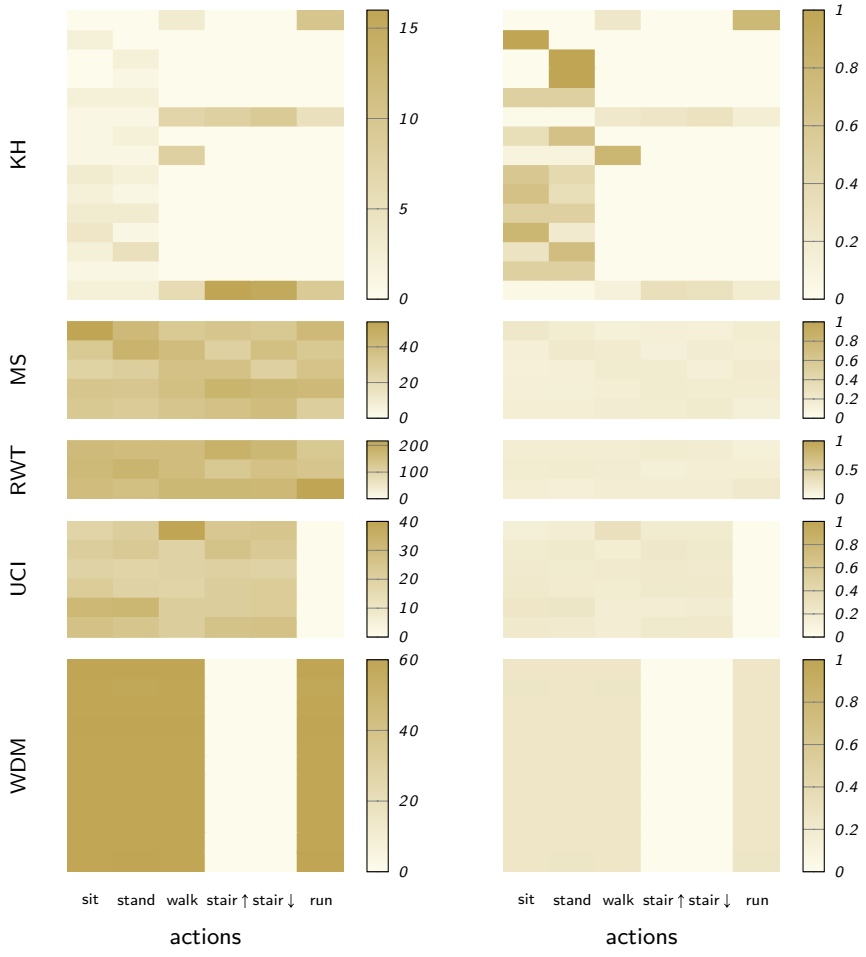
(a) Occurrences per user (b) Fraction of examples per user

Figure 1.7. Heatmaps of the DAGHAR training sets.



(a) Occurrences per user (b) Fraction of examples per user

Figure 1.8. Heatmaps of the DAGHAR validation sets.



(a) Occurrences per user (b) Fraction of examples per user

Figure 1.9. Heatmaps of the DAGHAR test sets.

2 CLASSIFICATION MODEL

This chapter describes a classification model for the HAR task in a supervised scenario utilizing deep learning techniques: the model integrates a recurrent network and a feedforward network, where the last output or internal state of the recurrent block serves as a feature descriptor for the feedforward classifier block.

2.1 BUILDING ELEMENTS

2.1.1 *MLP classifier*

Formally, an artificial neural network is specified in Def. 1.

Definition 1. *An* artificial neural network *is a directed graph*[1] $(\mathcal{U}, A)$ *with the following properties:*

A. *Each node* $u_i \in \mathcal{U}$ *is associated with a threshold value* ϑ_i*;*

B. *An edge* $(u_i, u_j) \in \mathcal{A}$ *between nodes* u_i *and* u_j *is associated with a real-valued parameter* w_{ij} *called a* weight*;*

C. *Each node* u_i *has an associated state variable* z_i*;*

D. *For each node* u_i*, a transfer function* $z_i = f(\vartheta_i, \{z_k, w_{ki} : k \neq i\})$ *determines the state* z_i *based on the threshold* ϑ_i*, the state* z_k *of connected nodes* $\{u_k\}$*, and the weights* w_{ki}*.*

Feed-forward networks

The simplest *deep networks* are called *multilayer perceptrons* (MLP) and consist of multiple layers of computational units known as *neurons*, each fully connected to those in the underlying layer (from which they receive inputs) and those in the upper layer (which they, in turn, influence) [29, p. 167]; an example is illustrated in Fig. 2.1.

MLP networks are generally arranged in layers, such that each unit receives input only from units in the immediately preceding layer.

Definition 2. *A network is defined as an* K-layer *network if its units are arranged in* $K + 1$ *levels* $\mathcal{L}_0 \dots \mathcal{L}_K$*, such that if unit* $u_i^{(a)}$ *in layer* $\mathcal{L}_a$ *is connected to unit* $u_j^{(b)}$ *in layer* $\mathcal{L}_b$*, then* $a < b$*. For a* K-layer *network in the strict sense, the additional condition* $b = a + 1$ *is required.*

1 A directed graph consists of an ordered pair $(\mathcal{U}, \mathcal{A})$, where $\mathcal{U} = \{u_1 \dots u_K\}$ represents the set of nodes, and $\mathcal{A} = \{(u_i, u_j) : u_i, u_j \in \mathcal{U}\}$ is the set of edges, understood as ordered pairs [2, p. 26].

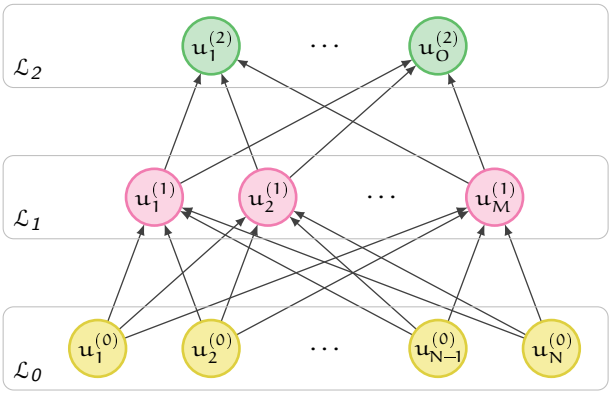


Figure 2.1. Example of an MLP network with one hidden layer (for visualization simplicity, the bias contribution has been omitted).

THE PERCEPTRON Before diving into MLP networks in detail, it is useful to recall the computational unit model that underlies them: the *perceptron*, as originally defined (Def. 3).

Definition 3. *A* perceptron *is a computational unit with a threshold* $\vartheta \in \mathbb{R}$ *that receives an input vector* $\mathbf{x} = [x_1 \dots x_N]^\top \in \mathbb{R}^N$*, computes a linear combination* $y = \mathbf{w}^\top \mathbf{x}$ *based on a* weight *vector* $\mathbf{w} = [w_1 \dots w_N]^\top \in \mathbb{R}^N$*, and finally produces an output* $f(\vartheta, \mathbf{x}, \mathbf{w})$ *that equals* 1 *if* $y > \vartheta$ *and* -1 *otherwise [11, p. 86].*

We can represent Def. 3 as a network with N input units and a single output unit $u_1^{(1)}$ (Fig. 2.2a): Each input unit $u_n^{(0)}$ provides its state x_n (which coincides with its input value) to the output unit according to the weight w_n. By adding a constant input $x_0 = 1$ (Fig. 2.2b), the threshold ϑ can be incorporated into the weight vector with an additional element $w_0 = -\vartheta$, allowing the perceptron definition to be compacted into the following formula[2] [11, p. 86]:

$$f(\mathbf{x}, \overline{\mathbf{w}}) = \operatorname{sgn}\left(\overline{\mathbf{w}}^\top \overline{\mathbf{x}}\right) \tag{2.1a}$$

$$\overline{\mathbf{x}} = [x_0\ x_1 \dots x_N]^\top = [1\ x_1 \dots x_N]^\top \quad \in \mathbb{R}^{N+1} \tag{2.1b}$$

$$\overline{\mathbf{w}} = [w_0\ w_1 \dots w_N]^\top = [-\vartheta\ w_1 \dots w_N]^\top \quad \in \mathbb{R}^{N+1} \tag{2.1c}$$

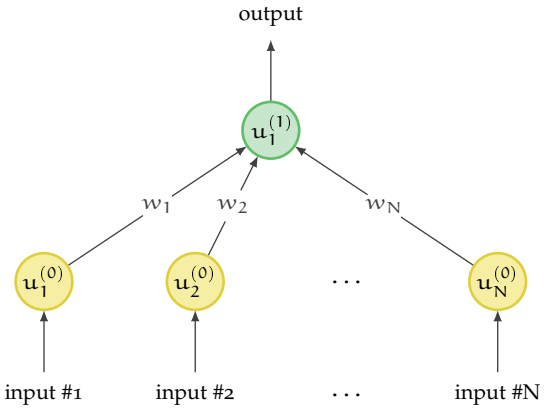
The additional weight w_0 is usually called the *bias*.[3] The sign function $\operatorname{sgn}(\varepsilon)$ constitutes the *activation function* of the perceptron.

Activation Functions

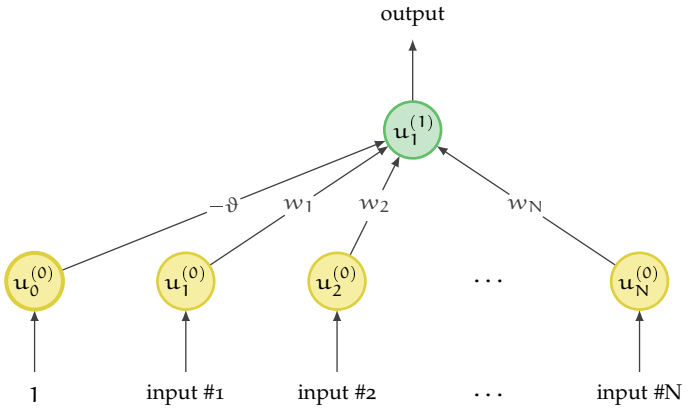
Activation functions determine whether a neuron should be activated by computing its weighted sum and adding an additional bias; the outputs of activation functions are called *activations*. Actually In MLP networks, the original sign function of the perceptron is replaced

2 $\operatorname{sgn}(\cdot)$ is the sign function.

3 This terminology stems from the fact that the quantity $\varepsilon = \overline{\mathbf{w}}^\top \overline{\mathbf{x}}$ is *biased* toward w_0 in the absence of inputs [4, p. 108].



(a) Perceptron as a 0-layer network.



(b) Extra node u_0, to compact the perceptron's transfer function.

Figure 2.2. Perceptron model.

by differentiable operators still introducing non-linearities in transforming input signals into outputs; below, we briefly recall the most common ones [29, pp. 171-174]:[4]

RECTIFIED LINEAR UNIT

$$\mathrm{ReLU}(x) = \max(0, x)$$

SIGMOID

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

HYPERBOLIC TANGENT

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(2x)}$$

4 The function $\mathrm{ReLU}(x)$ is theoretically non-differentiable at $x = 0$, but it can be made differentiable by explicitly defining its derivative at that point.

The Softmax Function

In the case of categorical data to be modeled as output, *one-hot encoding* is widely used, meaning that the network attempts to produce an output vector with as many components as there are C categories, where the component corresponding to the category of a particular instance is set to 1, and all other components are set to 0. However, treating classification as a regression problem with vector-valued outputs is still unsatisfactory for the following reasons [29, p. 127]:

- There is no guarantee that the sum of the outputs will be equal to 1, as expected for probabilities;
- There is no guarantee that the outputs will be non-negative, even if their sum equals 1, nor that they will not exceed 1.

A way to address these difficulties is to apply the exponential function to the final network output $\mathbf{u} = [u_1 \dots u_C]^\top$ to model class probabilities ($\mathbb{P}[i] \propto \exp u_i$) and normalize by their sum, ensuring they sum to 1, leading to the *softmax* function [29, p. 128].

$$\operatorname{softmax}(\mathbf{u}) = [f(u_1) \dots f(u_C)]^\top$$
$$f(u_i) = \frac{\exp u_i}{\sum_j \exp u_j}$$

2.1.2 *Recurrent Networks*

Recurrent models are widely used for processing sequential data, particularly in applications such as HAR, since temporal signals provide essential contextual information. By leveraging their ability to capture dependencies across time, they effectively model temporal patterns and improve classification accuracy. In this work, two kinds of recurrent models were considered: traditional Elman networks (usually referred to as simple recurrent neural networks) and the Long Short-Term Memory model (LSTM).

In case of *temporal* data, a recurrent model processes them by capturing dependencies between successive time steps. The observed data consist in a time sequence of values, which can take different forms:

- A sequence $x_1 \dots x_T$ of T samples collected into a vector $\mathbf{x} \in \mathbb{R}^T$;
- A sequence of T multidimensional vectors $\mathbf{x}_1 \dots \mathbf{x}_T$, where each $\mathbf{x}_t \in \mathbb{R}^d$, for $t = 1 \dots T$;
- A continuous signal that has been sampled.

The fundamental property of recurrent models is their ability to maintain an *internal state* $\mathbf{h}_t$ that evolves over time and depends on past inputs. This is typically expressed as:

$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, v_t)$$

where v_t represents a generic input data at time step t.

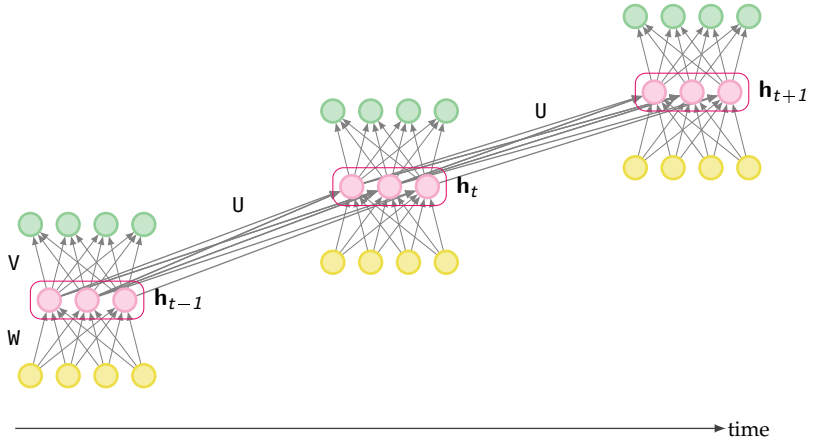


Figure 2.3. Simple RNN unfolded over time.

ELman Networks

A recurrent neural network is any network that contains a cycle in its connections, meaning that the value of a given unit depends directly or indirectly on its previous outputs as inputs. Specifically, Fig. 2.3 shows an example of an *Elman network*, (usually termed RNN for short), in which the internal state at time t is given by a linear combination of the current inputs and the previous internal state (the context), weighted by an appropriate (square) matrix $\mathtt{U}$ [5, § 8.1]

$$\begin{aligned}\mathbf{h}_t &= g(\mathtt{W}\mathbf{x}_t + \mathtt{U}\mathbf{h}_{t-1})\\ \mathbf{y}_t &= f(\mathtt{V}\mathbf{h}_t)\end{aligned}$$

In particular, we consider the case where $g(\cdot)$ is an activation function such as $\tanh(\cdot)$ or $\mathrm{ReLU}(\cdot)$, while $f(\cdot) = \mathrm{softmax}(\cdot)$ (so that the output models a probability distribution over the elements of $\mathbf{y}_t$ conditioned on the state $\mathbf{h}_t$).

STACKED RNNS Stacked RNNs comprise multiple recurrent layers, each taking the previous layer's output as input. This hierarchical design enables abstraction at various levels, typically enhancing performance over single-layer networks [5, § 8.4.1].

BIDIRECTIONAL RNNS So far, an RNN uses information from the left (previous) context to make its predictions at time t, and the hidden state at this instant represents everything the network has learned about the sequence up to that point. To also leverage context to the right of the current input, an RNN can be trained on the reversed input sequence. A bidirectional RNN thus combines two independent RNNs: one processes the input from start to end, while the other processes it from end to start. The internal representations of both networks are then concatenated into a single vector that captures both left and right context at each time step [5, § 8.4.2].

LSTM

The long short-term memory (LSTM) network [5, §8.5] is the most widely used extension of RNNs. LSTMs tackle context management by filtering out irrelevant information and integrating details likely needed for future decision-making.

The crucial aspect of solving both subproblems lies in learning how to regulate the context rather than encoding a predefined strategy directly into the architecture. LSTMs accomplish this by incorporating an explicit context layer into the architecture—alongside the standard recurrent hidden layer—and employing specialized neural units equipped with gates that regulate the flow of information into and out of the layers composing the network. These gates function through additional weights that sequentially act on the input, the previous hidden layer, and the previous context layers.

LSTM gates follow a common design principle: each consists of a feedforward layer, a sigmoid activation function, and an element-wise multiplication with the gated layer. The sigmoid function is chosen for its tendency to push outputs toward 0 or 1. Combined with element-wise multiplication, this creates an effect analogous to a binary mask (values close to 1 in the mask remain largely unchanged, while lower values are effectively suppressed).

THE FORGET GATE The first gate we consider is the *forget gate*. The purpose of this gate is to eliminate unnecessary information from the context. The forget gate computes a weighted sum of the hidden layer of the previous state and the current input and passes it through a sigmoid function. This mask is then element-wise multiplied by the context vector to remove information no longer needed in the context. Element-wise multiplication of two vectors (denoted by the operator $\odot$ and sometimes called the *Hadamard product*) results in a vector of the same size as the input vectors, where each element i is the product of the corresponding element in both input vectors:

$$\begin{aligned} \mathbf{f}_t &= \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t) \\ \mathbf{k}_t &= \mathbf{c}_{t-1} \odot \mathbf{f}_t \end{aligned}$$

The next task is to compute the actual information that needs to be extracted from the previous hidden state and the current inputs, following the same basic computation used for all recurrent networks.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t)$$

THE ADD GATE Next, we generate the mask for the *add gate* to select the information to be added to the current context.

$$\begin{aligned} \mathbf{i}_t &= \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t) \\ \mathbf{j}_t &= \mathbf{g}_t \odot \mathbf{i}_t \end{aligned}$$

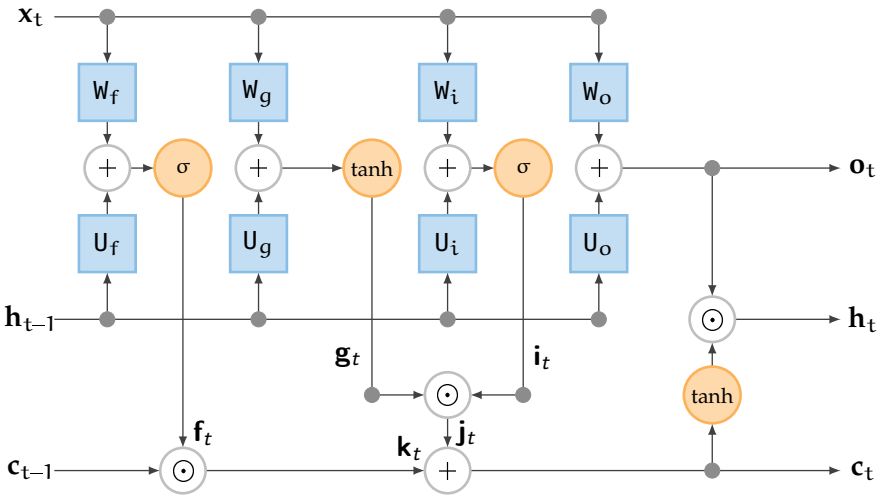


Figure 2.4. Computation graph for a single LSTM unit: the inputs for each unit consist of the current input, $\mathbf{x}$, the previous hidden state, $\mathbf{h}_{t-1}$, and the previous context, $\mathbf{c}_{t-1}$; the outputs are a new hidden state, $\mathbf{h}_t$, and an updated context, $\mathbf{c}_t$.

Then, we add it to the modified context vector to obtain the updated context vector as:

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

THE OUTPUT GATE The final gate is the *output gate*, which determines what information is needed for the current hidden state, as opposed to what should be retained for future decisions.

$$\mathbf{o}_t = \sigma(\mathtt{U}_o\mathbf{h}_{t-1} + \mathtt{W}_o\mathbf{x}_t)$$
$$\mathbf{h}_t = \mathbf{g}_t \odot \tanh \mathbf{c}_t$$

Fig. 2.4 illustrates the complete computation for a single LSTM unit. Given appropriate weights for the various gates, an LSTM takes as input the context level and the hidden state from the previous time step, along with the current input vector. It then generates the updated context and hidden vectors as output.

2.2 MODEL ASSEMBLY

The classifier model is structured as a processing pipeline in which the MLP classifier is cascaded to a recurrent network (either a simple RNN or an LSTM), providing the descriptor input to the MLP by its final internal layer or its final output (see also § 4.1.2).

Between the vector provided by the recurrent model and the input vector to the classifier, intermediate operations such as normalization and optional dropout are applied:

NORMALIZATION It operates on the final internal state of the model, normalizing each instance in the batch along all dimensions of

the hidden state related to that last timestep.[5] For each batch sample, assuming the internal state dimension is H, it calculates:

$$\hat{x_n} = \gamma \frac{x_n - \mu}{\sigma + \varepsilon} + \beta$$

$$\mu = \frac{1}{H} \sum_{n=1}^{H} x_m$$

$$\sigma = \sqrt{\frac{1}{H} \sum_{n=1}^{H} (x_n - \mu)^2}$$

where γ and β control *scaling* (γ) and *shifting* (β), learned during training, while $\varepsilon > 0$ is a small value added to prevent division by zero. The use of normalization generally serves to:

- Stabilize the descriptor by reducing its variance;
- Accelerate convergence, preventing extreme values in the descriptor from slowing optimization.

In this context, normalization helps stabilize the descriptors passed to the MLP.

DROPOUT Dropout applied to the descriptor randomly inserts zeros into some vector positions, disabling certain units with a specified probability. Similar to its application in recurrent or MLP models, dropout in this case acts as a regularization measure before the final classification step, slightly altering the input vector to make the model more robust to variations.

As descriptors produced by the recurrent network, we will use two different vectors: the final output of the network (*embedding from recurrent output*) or its last *internal* state, which, in the case of LSTM, is provided by the short-term memory vector.

COST FUNCTION Since this is a multi-class classification problem, the most suitable cost function for optimization is the *cross-entropy loss*:

$$L\left(\hat{\mathbf{y}}_n, \mathbf{y}_n\right)_{n=1}^{N} = \frac{\sum_{n=1}^{N} l(\hat{\mathbf{y}}_n, \mathbf{y}_n)}{N} \tag{2.3a}$$

$$l(\hat{\mathbf{y}}_n, \mathbf{y}_n) = -\sum_{c=1}^{C} y_{nc} \log \operatorname{softmax}(\hat{y}_{nc}) \tag{2.3b}$$

where $\hat{\mathbf{y}_n} = [\hat{y}_1 \dots \hat{y}_C]^\top \in \mathbb{R}^C$ represents the model's response (the output from the MLP classifier, in particular) after receiving the input example $\mathbf{x}_n \in \mathbb{R}^d$. Meanwhile, the ground truth vector $\mathbf{y}_n \in \mathbb{R}^C$ for a class-i example has all C components set to 0 except for the i-th component, which is set to 1.

5 In the context of recurrent neural networks (RNNs, LSTMs, etc.), a *timestep* represents a single point in time or an element within a data sequence. Each timestep corresponds to a unit of time or a position in the sequence that the recurrent model processes.

3 FEDERATED LEARNING

Federated learning is a machine learning technique where multiple entities collaboratively train a model while keeping data decentralized. Typically motivated by concerns such as privacy, minimization, and access rights, it is well suited to scenarios where large volumes of signals are acquired on personal devices capable of local processing, such as smartphones. Problems suited for federated learning exhibit the following characteristics [10]:

- Training on mobile device data offers a key advantage over relying on proxy data from the data center.
- Such data is either privacy-sensitive or large relative to model size, making centralized storage undesirable for training.

The learning task is handled by a federation of devices (*clients*) coordinated by a central *server*. Each client updates the global model using a local dataset that remains on the device—only the model update is shared [10].

Federated optimization has distinct properties differentiating it from typical distributed optimization problems [10]:

NON-IID DATA A client's training data reflects user's mobile activity, meaning local datasets may not align with the overall population distribution.

UNBALANCED DATA Some users engage with the service far more than others, causing variations in local training data volume.

HIGHLY DISTRIBUTED PROBLEM The number of participating clients typically exceeds the average number of examples per client.

LIMITED COMMUNICATION Mobile devices are often offline or connected via slow or costly networks.

Beyond these factors, practical challenges arise: client datasets evolve as data are added or deleted, and device availability is intricately linked to local data distribution.[1] Some clients may also fail to respond or transmit corrupted updates.

3.1 THE "FEDERATED AVERAGING" ALGORITHM

In [10], a synchronous update scheme is analyzed that proceeds in communication cycles, addressing key challenges regarding client availability, unbalanced data, and non-IID data:

1 For instance, U.S. and British users' phones tend to connect at different times.

- There is a fixed set of K clients, each with a fixed local dataset;
- At the start of each cycle, a random fraction C of clients is selected, and the server transmits the algorithm's current global state to them (e.g., model parameters).
- Only a fraction of clients is selected for efficiency reasons;[2]
- Each selected client performs local computations using the global state and its local dataset, then sends an update to the server;
- The server applies the updates to its state and repeats the cycle.

The procedure is described using pseudo-code in Alg. 1; specifically, it details the operations performed on both the client side (each of which performs several steps of *stochastic gradient descent* on its local loss function L_k) and the server side (which updates the central model based on the updates received from clients) to minimize the overall loss function L. For the scenario analyzed here, recall Eq. (2.3) for L.

3.2 THE FLOWER LIBRARY

Flower [1] is a Python framework for implementing distributed systems via federated learning in a client-server architecture:

- The server coordinates the federation process, gathers model updates from clients, and aggregates parameters to generate a shared global model, supporting predefined strategies such as Federated Averaging (FedAvg);
- Clients represent distributed nodes that perform training on their own data without transferring raw datasets.

3.2.1 *Supporting Components*

Having data available in separate datasets per user (e.g., `user_n.csv` with $n = 1 \ldots N$), let `load_datasets(user:int, batch_size=256)` be the function that returns data loaders for the training, validation, and test sets of each user.

Suppose we have a subclass of `torch.nn.Module` defining the model to be learned, called `Net`. Then, two functions are required:

- `train(net:Net, train_loader:DataLoader, epochs:int)`, which trains `net` on data from `train_loader` over `epochs` iterations;
- `test(net:Net, data_loader:DataLoader)`, which evaluates `net` using `data_loader`, either from the validation or test set.

2 n [10], experiments show diminishing returns beyond a certain number of clients.

Algorithm 1: Federated Averaging Algorithm

Input:
R: # of rounds;
γ: fraction of clients performing computations at each round;
K: # of clients; B: batch size per client;
E: # of training epochs per client;
η: learning rate;
$\mathbf{w}_0$: initial model weights on server and clients
Output: Model weights $\mathbf{w}$ on the server

On server
for $t = 1, \dots R$ **do**
  $m \leftarrow \max(\gamma \cdot K, 1)$
  $\mathcal{S}_t \leftarrow$ (random set of m clients)
  for $k \in \mathcal{S}_t$ *in parallel* **do**
    $\mathbf{w}_{t+1}^{(k)} \leftarrow \texttt{ClientUpdate}(k, \mathbf{w}_t)$
    $n_k \leftarrow |\mathcal{P}_k|$ (# of examples $\mathcal{P}_k$ on client k)
  end
  $n \leftarrow \sum_{k=1}^{m} n_k$
  $\mathbf{w}_{t+1} \leftarrow \frac{1}{n} \sum_{k=1}^{m} n_k \mathbf{w}_{t+1}^{(k)}$
end
On client (procedure `ClientUpdate`$(k, \mathbf{w})$)
$\mathcal{B} \leftarrow$ split local data $\mathcal{P}_k$ into batches of size B
for $i = 1$ ***to*** E **do**
  for *batch* $b \in \mathcal{B}$ **do**
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L_k(\mathbf{w}; b)$
  end
end
return $\mathbf{w}$ to server

Due to the client-server mechanism,[3] two *helper* functions are needed to handle parameter exchange: `set_parameters` (to update the local model with received parameters) and `get_parameters` (to retrieve the updated parameters).

Flower creates a `ClientApp` instance per user for executing client-side code and a `ServerApp` object for server-side operations.

3.2.2 *ClientApp*

`ClientApp` is the entry point that a running Flower client uses to invoke the code (as defined, for example, in `FlowerClient.fit()`).

3 We recall that the server sends global model parameters to the client, which updates its local model accordingly. The client trains the model on its data (modifying local parameters) and sends back either the updated parameters or just the gradients.

To define `ClientApp`, a subclass of `flwr.client.NumPyClient`, which we will refer to as `FlowerClient`, can be declared.[4] In order to implement `NumPyClient` class, `FlowerClient` must contain the methods `get_parameters()`, `fit()`, and `evaluate`:

- `get_parameters()` returns the current local model parameters;
- `fit()` receives the model parameters from the server, trains the model on local data, and returns the updated model parameters to the server;
- `evaluate()` receives the model parameters from the server, evaluates the model on local data, and returns the evaluation result to the server.

The `FlowerClient` class defines how local training/evaluation will be performed and allows Flower to invoke training via `fit()` and local evaluation via `evaluate()`.[5]

To allow Flower to create clients when needed, a function that creates an instance of `FlowerClient` on demand must be implemented; this function is typically named `client_fn()`. Flower calls `client_fn()` whenever it needs an instance of a particular client to invoke `fit()` or `evaluate()` (these instances are usually discarded after use, so they should not maintain any local state).

In federated learning with Flower, clients are identified by an ID denoted as `partition-id`, which is contained in the dictionary `node_config` within the `Context` object, holding information that persists throughout each training cycle. This identifier is used to load different partitions of local data for different clients.

Now, the instruction `client = ClientApp(client_fn=client_fn)` allows the creation of an instance of `ClientApp`. The code in List. 3.1 summarizes the discussed implementation.[6]

3.2.3 *ServerApp*

`ServerApp` is the entry point that Flower uses to invoke all server-side code, such as the *strategy* encapsulating the federated learning algorithm (in this case, `FedAvg`).

Similar to `ClientApp`, an object `ServerApp` must be created using the utility function `server_fn()`, within which an instance of

4 More generally, the parent class is `flwr.client.Client`; `flwr.client.NumPyClient` simplifies the implementation by removing redundant steps.

5 Flower provides a way to send configuration values from the server to clients using a dictionary `config`, passed as an argument to `fit()` and `evaluate()`, through which the desired working configuration can be communicated to the client. To pass these dictionaries to the client, they must be provided as parameters to the `strategy` object representing the learning strategy (see § 3.2.3).

6 The variable `NUM_CLIENTS` will be needed later (List. 3.2 and List. 3.3), but it is declared in List. 3.1 for convenience and subject relevance.

`ServerConfig` is passed to define the number of federated learning rounds (`num_rounds`). Additionally, the previously created `strategy` object is provided: `server_fn()` returns a `ServerAppComponent` object containing the settings that define the behavior of `ServerApp` (List. 3.2).

If a configuration dictionary needs to be passed to clients, for example, to execute the `fit()` function, the declaration of the `strategy` object in List. 3.2 must include the entry `on_fit_config_fn=fit_config`, encapsulating the desired execution mode.

Listing 3.1: Code to generate an instance of `ClientApp`.

```
NUM_CLIENTS = 10
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

class FlowerClient(NumPyClient):
  def __init__(self, partition_id, net, train_loader,
      valid_loader):
    self.partition_id = partition_id
    self.net = net
    self.trainloader = train_loader
    self.valloader = valid_loader

  def get_parameters(self, config):
    print(f"[Client {self.partition_id}] get_parameters")
    return get_parameters(self.net)

  def fit(self, parameters, config):
    print(f"[Client {self.partition_id}] fit, config: {config}")
    set_parameters(self.net, parameters)
    train(self.net, self.trainloader, epochs=1)
    return get_parameters(self.net), len(self.trainloader), {}

  def evaluate(self, parameters, config):
    print(f"[Client {self.partition_id}] evaluate, config:
        {config}")
    set_parameters(self.net, parameters)
    loss, accuracy = test(self.net, self.valloader)
    return float(loss), len(self.valloader), {"accuracy":
        float(accuracy)}

def client_fn(context: Context) -> Client:
  net = Net().to(DEVICE)

  # Read the node_config to fetch data partition associated with
      this node
  partition_id = context.node_config["partition-id"]
  train_loader, valid_loader, _ = load_datasets(partition_id)
  return FlowerClient(partition_id, net, train_loader,
      valid_loader).to_client()

# Create the ClientApp
client = ClientApp(client_fn=client_fn)
```

Central Model Initialization

By default, Flower initializes the central model by requesting initial parameters from a random client. However, in List. 3.2, an instance of `Net` is created, enabling direct parameter passing to the strategy.

Model Evaluation (Server-Side)

Flower can evaluate the aggregated model either server-side (centralized) or client-side (federated).

CENTRALIZED EVALUATION Works similarly to evaluation in centralized machine learning: the model can be assessed after each training cycle without sending it to clients.

FEDERATED EVALUATION Does not require a centralized dataset and allows models to be evaluated on a broader dataset, often yielding more realistic evaluation results. However, once client-side evaluation begins, the evaluation dataset may change across consecutive training cycles if clients are not always available. Additionally, each client's dataset may evolve over multiple cycles.

Federated evaluation is implemented via the `evaluate()` method in `FlowerClient`. To evaluate aggregated model parameters on the server side, simply define an `evaluate()` function and pass it to the `FedAvg` strategy (List. 3.2).

The `evaluate()` function in List. 3.2 takes the following inputs:

- `server_round:int` is an integer representing the communication round between server and clients.
- `parameters:NDArrays` contains the model parameters as multi-dimensional `NDArrays` (handled with `numpy`);
- `config:Dict[str, Scalar]` is a dictionary with numerical values and string keys, containing additional configuration settings for the evaluation process, such as learning rate or other hyperparameters.

The output type `Optional[Tuple[float, Dict[str, Scalar]]]` means `evaluate()` may return:

- `None`, if there is no result to return;
- a tuple, consisting of a float (representing the loss) and a dictionary (with numerical values and string keys) containing additional information and metrics such as accuracy.

3.2.4 *Simulation Execution*

Finally, as the last step, it is necessary to specify the resources for each client before running the simulation (List. 3.3).

Listing 3.2: Code to generate an instance of `ServerApp`.

```
# Create an instance of the model and get the parameters
params = get_parameters(Net())

# The ‘evaluate‘ function will be called by Flower after every
    round
def evaluate(
    server_round: int,
    parameters: NDArrays,
    config: Dict[str, Scalar],
  ) -> Optional[Tuple[float, Dict[str, Scalar]]]:

  net = Net().to(DEVICE)
  _, _, testloader = load_datasets(0)
  # Update model with the latest parameters
  set_parameters(net, parameters)
  loss, accuracy = test(net, testloader)
  return loss, {"accuracy": accuracy}

def server_fn(context: Context) -> ServerAppComponents:
  # Create the FedAvg strategy
  strategy = FedAvg(
    fraction_fit=0.3,
    fraction_evaluate=0.3,
    min_fit_clients=3,
    min_evaluate_clients=3,
    min_available_clients=NUM_CLIENTS,
    initial_parameters=ndarrays_to_parameters(params),
    evaluate_fn=evaluate, # Pass the evaluation function
  )
  # Configure the server for 3 rounds of training
  config = ServerConfig(num_rounds=3)
  return ServerAppComponents(strategy=strategy, config=config)

# Create the ServerApp
server = ServerApp(server_fn=server_fn)
```

Listing 3.3: Instructions for launching the simulation.

```
# Specify the resources each client needs. If set to none,
# each client will be allocated 2x CPU and 0x GPUs
backend_config = {"client_resources": None}
if DEVICE.type == "cuda":
  backend_config = {"client_resources": {"num_gpus": 1}}

# Run simulation
run_simulation(
  server_app=server,
  client_app=client,
  num_supernodes=NUM_CLIENTS,
  backend_config=backend_config,
)
```

4

EXPERIMENTS

Experiments were carried out in both a centralized setup and a federated learning scenario, allowing for a comparative analysis of the two approaches.

4.1 CENTRALIZED APPROACH

Two types of models are tested: one employing a simple RNN and another using an LSTM to extract the embedding vector to be passed as input to the MLP block:

- For the simple RNN, a single layer with 64 or 128 neurons;
- Two layers (of equal size, also with 64 or 128 elements) with a *bidirectional* LSTM.

The cascaded MLP network consists of 3 layers, with the internal layer containing 64 or 128 neurons.

4.1.1 *Learning Rate Selection*

To robustly estimate an optimal learning rate, an approach combining several techniques was adopted:

- First, the *Learning Rate Range Test*[1] [16, pp. 53-54][21] was performed $N_T = 1000$ times over the interval $[r_{min}, r_{max}]$;
- The results from these runs were averaged to reduce variability, as any single realization was very noisy (Fig. 4.1);
- Next, the average loss curve was smoothed using a Savitzky-Golay filter [18, pp. 766-772], applying a third-degree polynomial evaluated over a neighborhood of radius 20;
- Finally, the derivative of this curve was computed using a third-order central difference approximation [23, pp. 311-312]:

$$f'_n = \frac{-f_{n+2} + 8f_{n+1} - 8f_{n-1} + f_{n-2}}{12h} + O(h^4)$$

where the sampling step h (actually here not strictly necessary) in this case is evaluated on a logarithmic scale given the number N_b of batches covering the entire dataset within an epoch:

$$h = \frac{\log_{10} r_{max} - \log_{10} r_{min}}{N_b} = \log_{10} \left(\frac{r_{max}}{r_{min}} \right)^{(1/N_b)}$$

1 In the Learning Rate Range Test, training starts at r_{min} and gradually increases per batch until reaching r_{max}, computing the loss throughout.

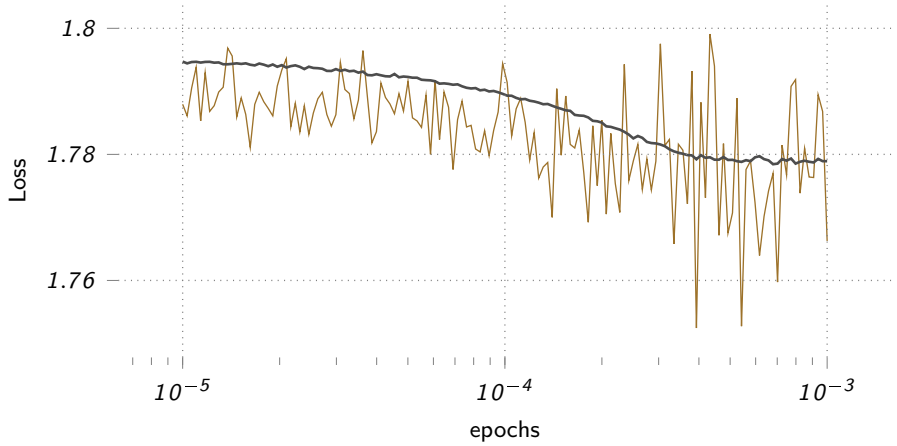


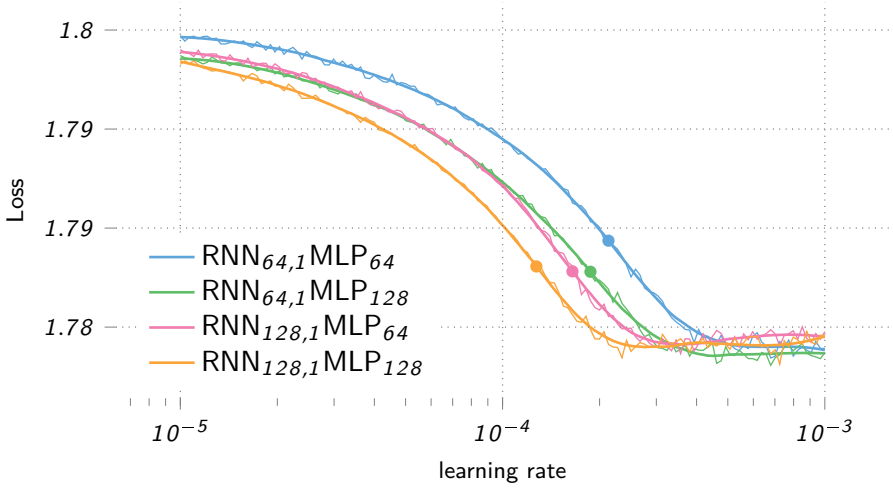
Figure 4.1. Single realization of loss with a variable learning rate for a simple RNN with 64 internal neurons and the same number for the MLP; in black, the average of $N_T = 1000$ realizations.

- The point with the steepest downward slope is selected as an estimate of the optimal learning rate.

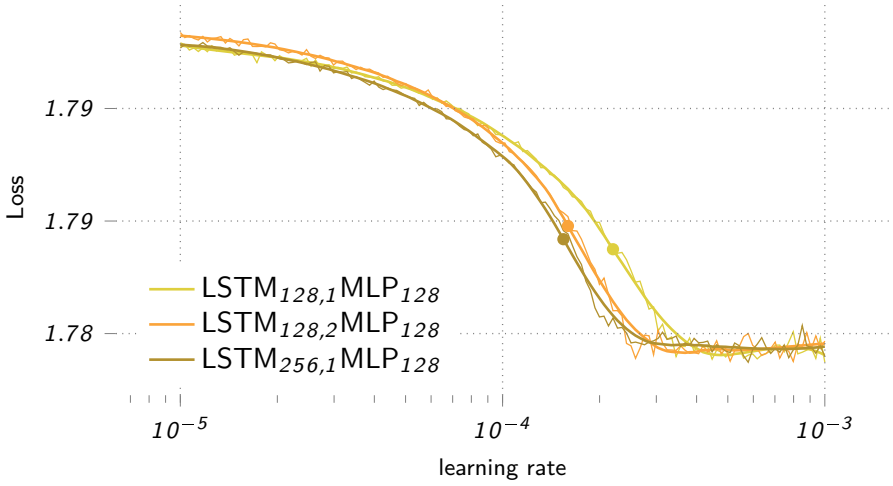
In all tests, a batch size of 256 examples is used, with $r_{min} = 10^{-5}$ and $r_{max} = 10^{-3}$. Tab. 4.1 reports the learning rates obtained using the described procedure, employing an MLP classifier cascaded to a simple recurrent network with a single hidden layer or a bidirectional LSTM with 2 hidden layers, varying the number of internal neurons. Fig. 4.2 illustrates the loss trends as a function of the learning rate across various configurations.

Table 4.1. Learning rates for various HAR model configurations; the columns indicating the number of internal neurons per layer ("internal dim.") also specify the number of hidden layers in parentheses.

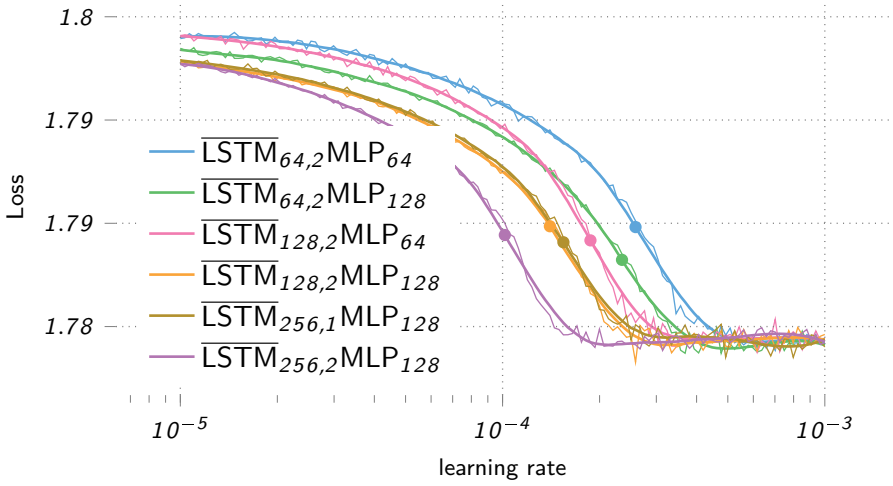
	internal dim.			
RNN type	RNN	MLP	code	learn. rate $\cdot 10^4$
Simple RNN	64(1)	64(1)	$RNN_{64,1}MLP_{64}$	2.131
	64(1)	128(1)	$RNN_{64,1}MLP_{128}$	1.874
	128(1)	64(1)	$RNN_{128,1}MLP_{64}$	1.647
	128(1)	128(1)	$RNN_{128,1}MLP_{128}$	1.273
LSTM	128(1)	128(1)	$LSTM_{128,1}MLP_{128}$	2.201
	128(2)	128(1)	$LSTM_{128,2}MLP_{128}$	1.595
	256(1)	128(1)	$LSTM_{256,1}MLP_{128}$	1.595
Bidir. LSTM	64(2)	64(1)	$\overline{LSTM}_{64,2}MLP_{64}$	2.131
	64(2)	128(1)	$\overline{LSTM}_{64,2}MLP_{128}$	2.586
	128(2)	64(1)	$\overline{LSTM}_{128,2}MLP_{64}$	1.874
	128(2)	128(1)	$\overline{LSTM}_{128,2}MLP_{128}$	1.402
	256(1)	128(1)	$\overline{LSTM}_{256,1}MLP_{128}$	1.595
	256(2)	128(1)	$\overline{LSTM}_{256,2}MLP_{128}$	1.016



(a) RNN + MLP



(b) LSTM + MLP



(c) LSTM bidirezionale + MLP

Figure 4.2. Average over $N_T = 1000$ realizations of the loss as the learning rate varies (thin line), along with smoothing obtained via Savitzky-Golay (thick line), highlighting the point of maximum slope.

Table 4.2. Accuracy for the models in Tab. 4.1, using the learning rates indicated therein.

	accuracy %				
architecture	training	validation	test	epoch	details
$RNN_{64,1}MLP_{64}$	54.1	54.7	53.5	187	Fig. 4.3
$RNN_{64,1}MLP_{128}$	52.8	53.0	52.0	188	Fig. 4.4
$RNN_{128,1}MLP_{64}$	41.8	43.1	42.7	192	Fig. 4.5
$RNN_{128,1}MLP_{128}$	44.4	44.6	44.6	192	Fig. 4.6
$LSTM_{128,1}MLP_{128}$	73.8	69.3	68.1	167	Fig. 4.7
$LSTM_{128,2}MLP_{128}$	76.2	71.0	68.3	149	Fig. 4.8
$LSTM_{256,1}MLP_{128}$	77.3	71.6	71.3	158	Fig. 4.9
$\overline{LSTM}_{64,2}MLP_{64}$	72.2	68.5	67.3	188	Fig. 4.10
$\overline{LSTM}_{64,2}MLP_{128}$	75.8	**72.2**	70.8	193	Fig. 4.11
$\overline{LSTM}_{128,2}MLP_{64}$	76.7	69.6	68.5	171	Fig. 4.12
$\overline{LSTM}_{128,2}MLP_{128}$	72.3	67.6	64.9	159	Fig. 4.13
$\overline{LSTM}_{256,1}MLP_{128}$	73.7	70.6	67.9	118	Fig. 4.14
$\overline{LSTM}_{256,2}MLP_{128}$	75.8	70.2	69.1	168	Fig. 4.15

4.1.2 *Centralized Trainings*

Tab. 4.2 reports the accuracy on the training set, validation set, and test set for the models in Tab. 4.1 (using the learning rates indicated therein), at the point of minimum loss for the validation set over a horizon of 200 epochs. Fig. 4.3–4.15 illustrate the loss and accuracy trends during training: in almost all cases, early signs of overfitting appear within 200 epochs.

The best model was obtained using a bidirectional LSTM with two internal layers of 64 neurons each, followed by an MLP with a single layer of 128 neurons. This model achieved an accuracy of 72.2% on the validation set (75.8% on the training set and 70.8% on the test set).

Tab. 4.3 summarizes the learning rates obtained using the hidden layer of a recurrent model (either a simple RNN or an LSTM) as the descriptor input to the MLP classifier, along with the accuracy observed over 200 epochs. The results are consistent with those in Tab. 4.1 and Tab. 4.2, confirming that the best model is a bidirectional LSTM with 2 hidden layers of 64 neurons each, followed by an MLP block with a hidden layer of 128 units.

Fig. 4.16 presents the confusion matrices for the training set, validation set, and test set as heatmaps. In Fig. 4.16–4.19, the heatmaps for the confusion matrices are "exploded," separating examples according to their original repositories.

Since accuracy is slightly improved compared to using the final output of the recurrent block (73.3% on the validation set and 71.4% on the test set), this configuration will be used in the federated model.

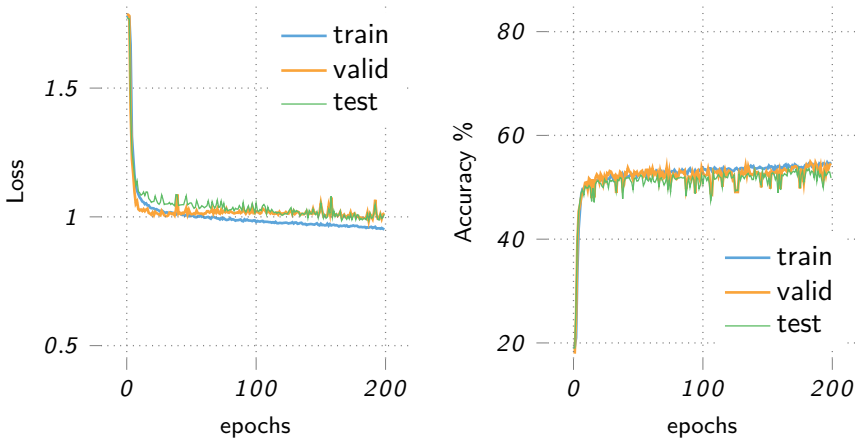


Figure 4.3. $\mathrm{RNN}_{64,1}\mathrm{MLP}_{64}$

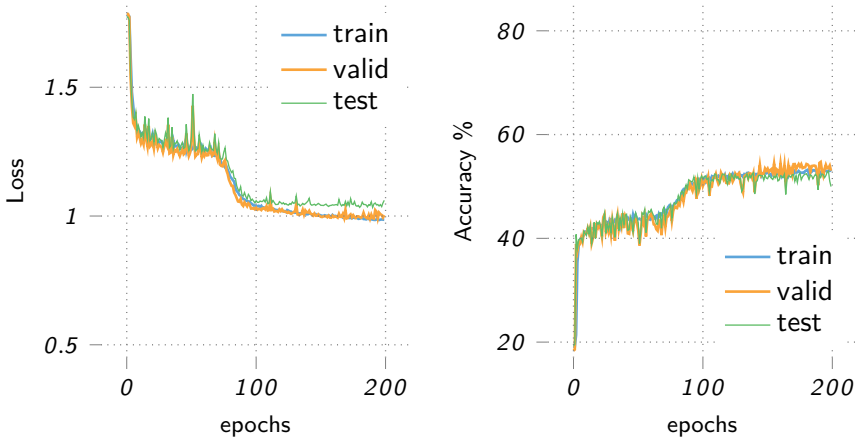


Figure 4.4. $\mathrm{RNN}_{64,1}\mathrm{MLP}_{128}$

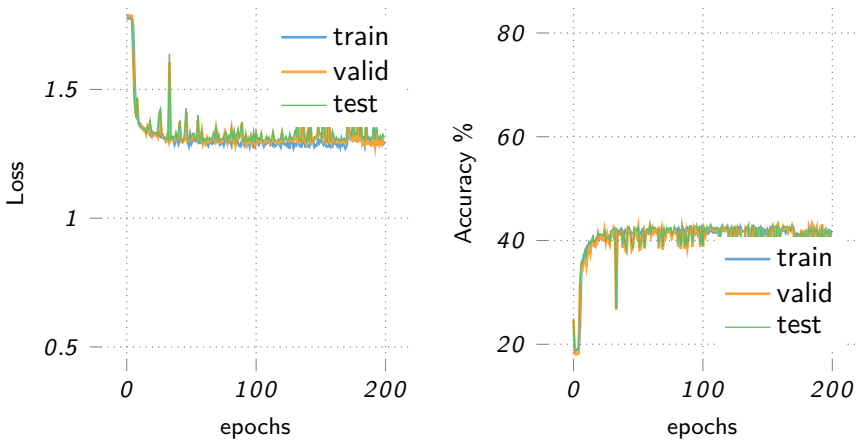


Figure 4.5. $\mathrm{RNN}_{64,1}\mathrm{MLP}_{128}$

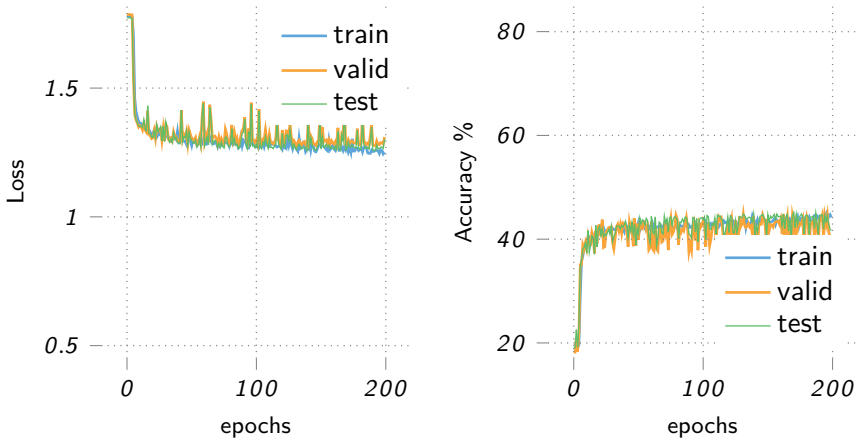


Figure 4.6. $\text{RNN}_{128,1}\text{MLP}_{128}$

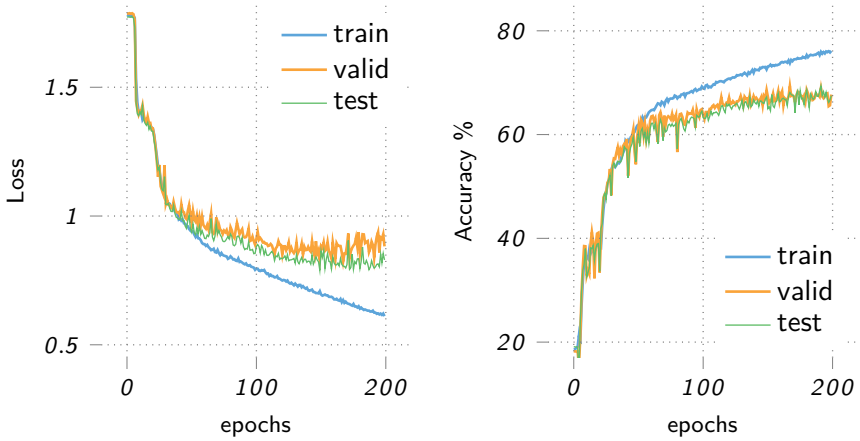


Figure 4.7. $\text{LSTM}_{128,1}\text{MLP}_{128}$

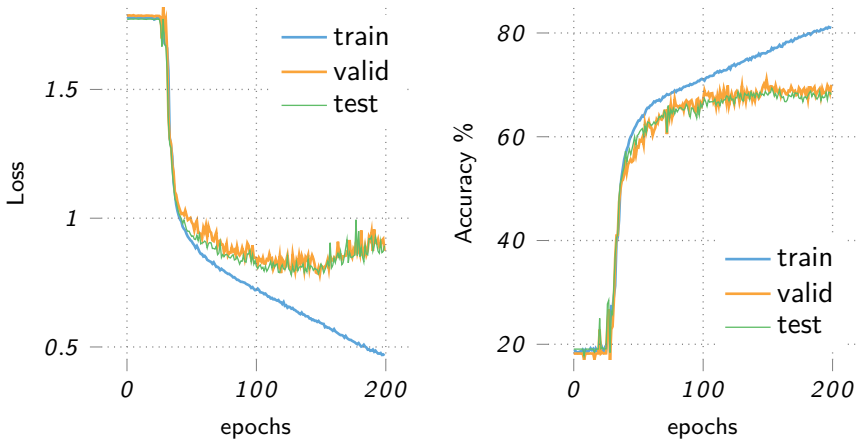


Figure 4.8. $\text{LSTM}_{128,2}\text{MLP}_{128}$

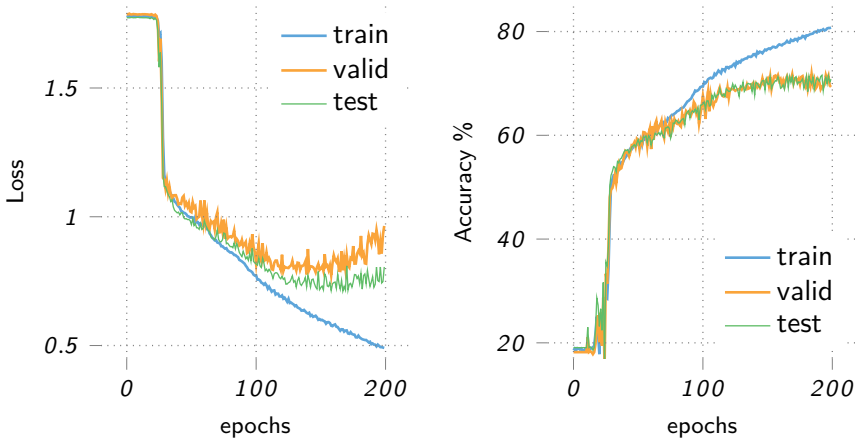


Figure 4.9. $\text{LSTM}_{256,1}\text{MLP}_{128}$

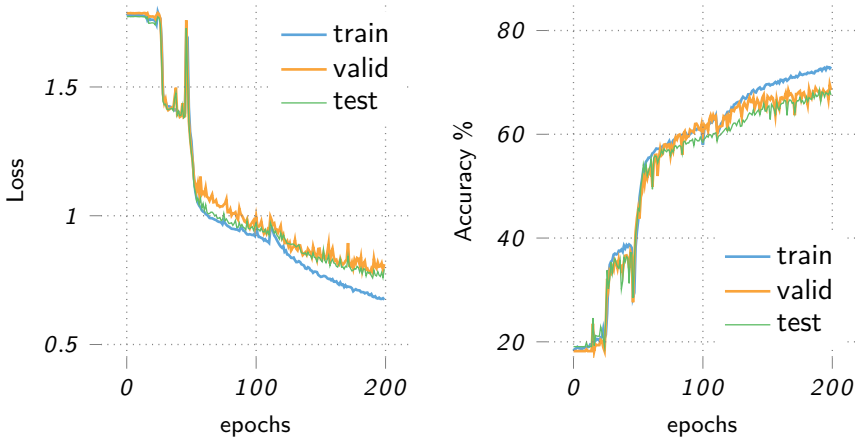


Figure 4.10. $\overline{\text{LSTM}}_{64,2}\text{MLP}_{64}$

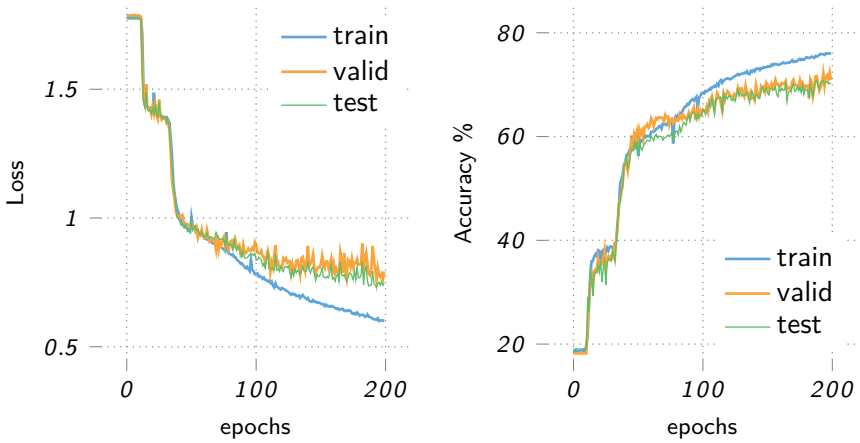


Figure 4.11. $\overline{\text{LSTM}}_{64,2}\text{MLP}_{128}$

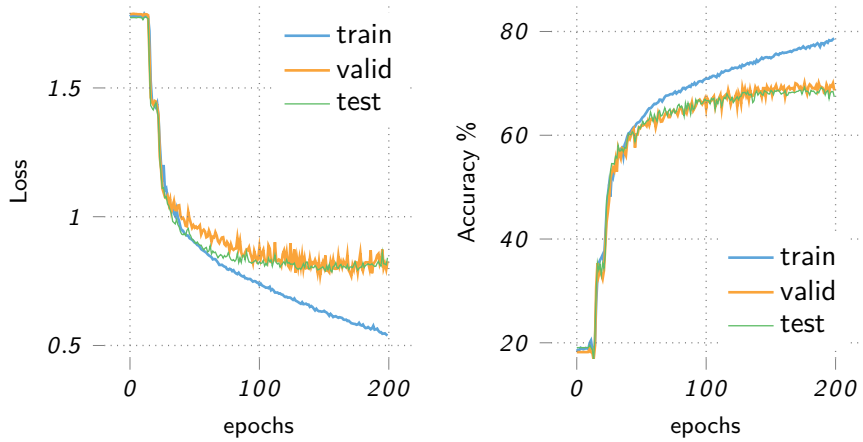


Figure 4.12. $\overline{\text{LSTM}}_{128,2}\text{MLP}_{64}$

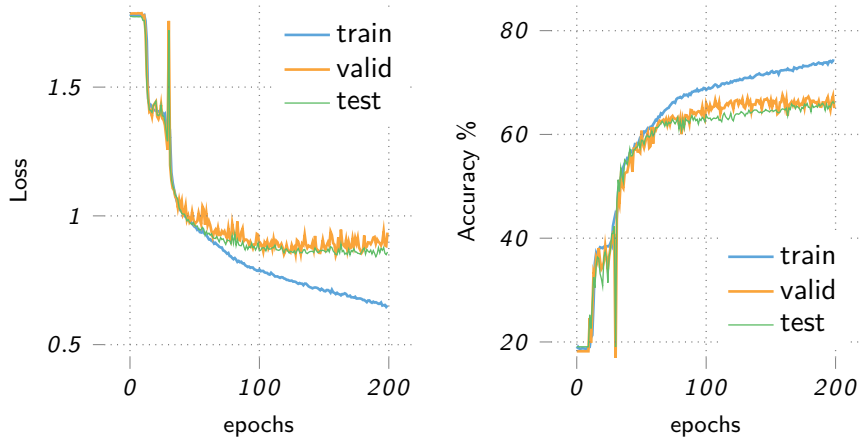


Figure 4.13. $\overline{\text{LSTM}}_{128,2}\text{MLP}_{128}$

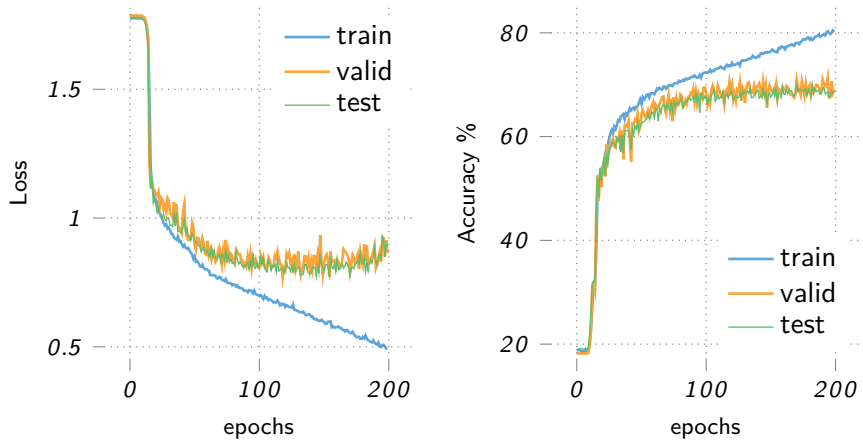


Figure 4.14. $\overline{\text{LSTM}}_{256,1}\text{MLP}_{128}$

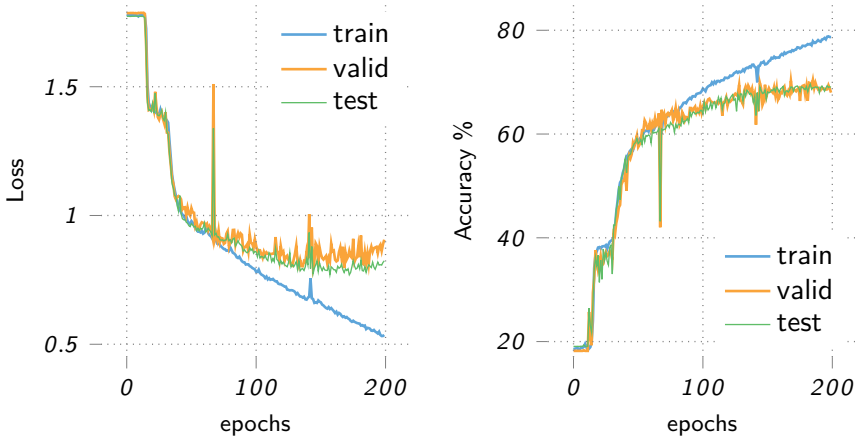


Figure 4.15. $\overline{\text{LSTM}}_{256,2}\text{MLP}_{128}$

Table 4.3. Learning rate and accuracy for various HAR model configurations taken from Tab. 4.1 and Tab. 4.2, using the final hidden layer.

		accuracy %			
code	learn. rate $\cdot 10^4$	train.	valid.	test	epoch
$\text{RNN}_{64,1}\text{MLP}_{64}$	2.131	54.1	54.7	53.5	188
$\text{RNN}_{64,1}\text{MLP}_{128}$	1.874	52.8	53.0	52.0	189
$\text{RNN}_{128,1}\text{MLP}_{64}$	1.647	41.8	43.1	42.7	193
$\text{RNN}_{128,1}\text{MLP}_{128}$	1.273	44.4	44.6	44.6	193
$\text{LSTM}_{128,1}\text{MLP}_{128}$	2.201	73.8	69.3	68.1	168
$\text{LSTM}_{128,2}\text{MLP}_{128}$	1.595	76.2	71.0	68.3	150
$\text{LSTM}_{256,1}\text{MLP}_{128}$	1.595	77.3	71.6	71.3	159
$\overline{\text{LSTM}}_{64,2}\text{MLP}_{64}$	2.131	76.2	70.2	66.8	168
$\overline{\text{LSTM}}_{64,2}\text{MLP}_{128}$	2.201	79.4	**73.3**	71.4	168
$\overline{\text{LSTM}}_{128,2}\text{MLP}_{64}$	1.647	78.4	70.4	69.5	182
$\overline{\text{LSTM}}_{128,2}\text{MLP}_{128}$	1.402	77.6	69.9	67.8	190
$\overline{\text{LSTM}}_{256,1}\text{MLP}_{128}$	1.233	69.7	65.4	64.8	194
$\overline{\text{LSTM}}_{256,2}\text{MLP}_{128}$	0.953	71.1	66.9	65.7	87

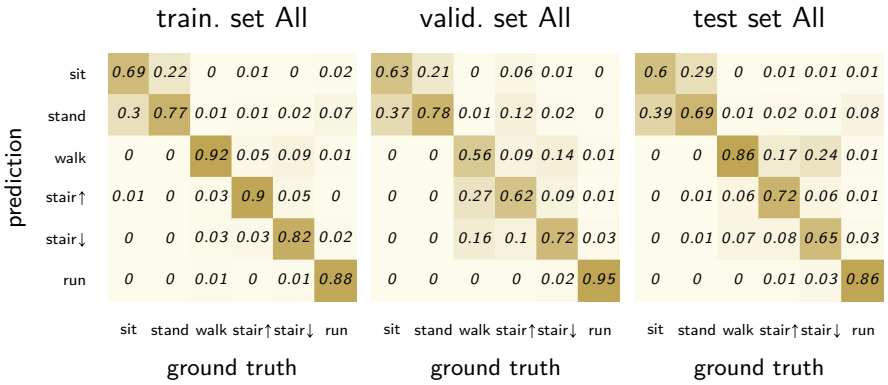


Figure 4.16. Heatmaps for confusion matrices separated by training set, validation set, and test set with aggregated repositories.

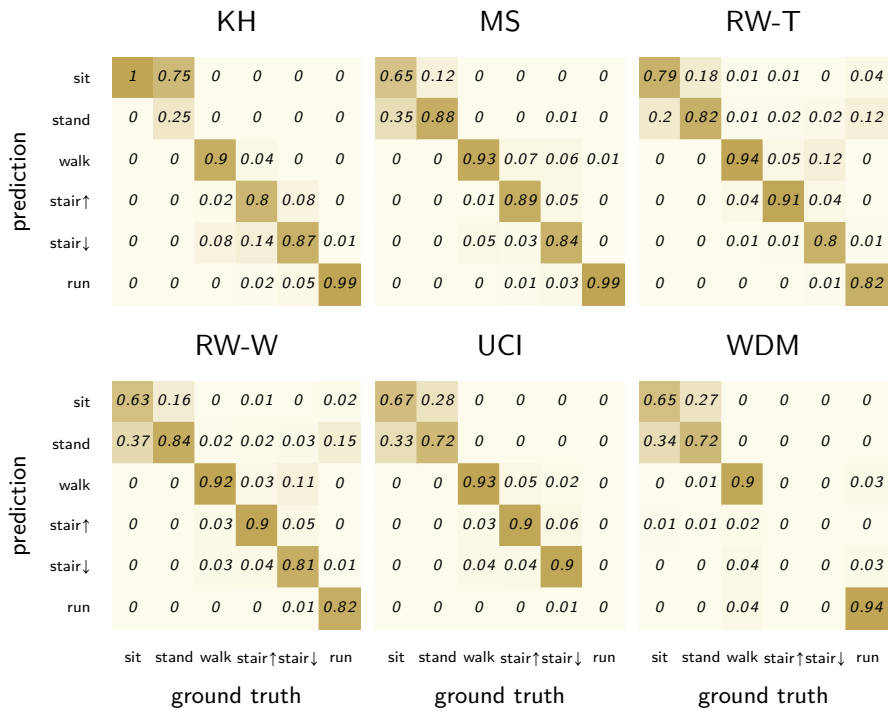


Figure 4.17. Confusion matrices (heatmaps) for the training set, by source.

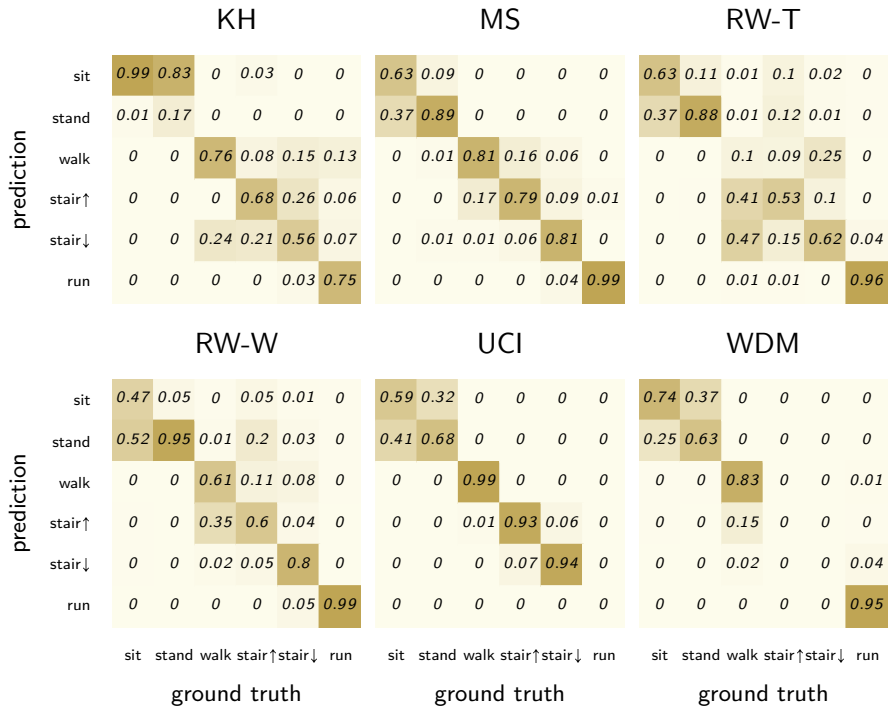


Figure 4.18. Confusion matrices (heatmaps) for the validation set, by source.

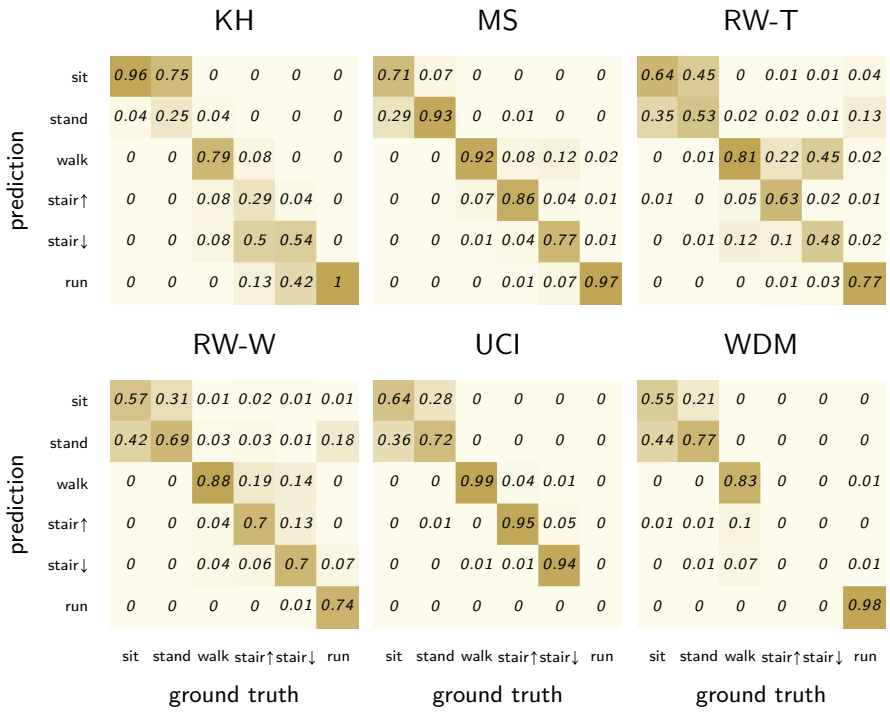


Figure 4.19. Confusion matrices (heatmaps) for the test set, by source.

The previous results were obtained without employing specific techniques to mitigate overfitting, notably the omission of dropout and layer normalization. To enhance the model's performance, these methods were incorporated and tested to determine the optimal configuration so far:

- Dropout is applied within the internal states of the LSTM block and to the feature vector input to the MLP classifier;
- Layer normalization is applied exclusively to the feature vector.

Table 4.4 shows the accuracy results for various dropout levels (combined with layer normalization as described), with the best performance at a dropout level of 0.4: Compared to Table 4.2, this setup improves accuracy for both the validation set (76.5% vs. 73.3%) and the test set (72.1% vs. 71.4%). Fig. 4.20 illustrates the progression of loss and accuracy during training (epochs range $[0, 200]$), while Figs. 4.21–4.24 present the confusion matrices as heatmaps.

As observed in the previous confusion matrices (Figs. 4.16-4.19), the most challenging activities are 'sit' and 'stand,' as distinguishing between them is quite difficult. This is further confirmed by Fig. 4.25, which presents a 2D projection of the embedding vectors generated by the LSTM module using the t-SNE algorithm [26].[2]

2 The t-SNE algorithm (t-distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique designed for visualizing high-dimensional data. It models affinities between data points as probabilities in the original space using Gaussian

Table 4.4. Accuracy by varing dropout value (and with layer normalization) for the model $\overline{\text{LSTM}}_{64,2}\text{MLP}_{128}$.

	accuracy %			
dropout	train.	valid.	test	epoch
0.0	71.4	71.4	66.7	47
0.1	74.9	70.7	70.0	83
0.2	76.0	72.7	71.5	117
0.3	71.9	68.2	67.6	182
0.4	81.9	**76.5**	72.1	195
0.5	74.1	71.7	70.8	99
0.6	76.4	75.1	71.0	153
0.7	71.8	72.6	68.2	144
0.8	75.5	70.8	70.9	186
0.8	75.5	70.8	70.9	186
0.9	62.2	61.2	60.1	149

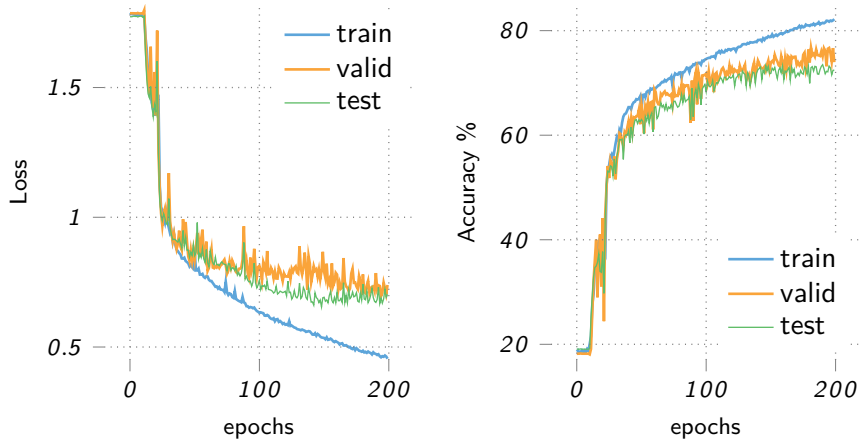


Figure 4.20. $\overline{\text{LSTM}}_{64,2}\text{MLP}_{128}$

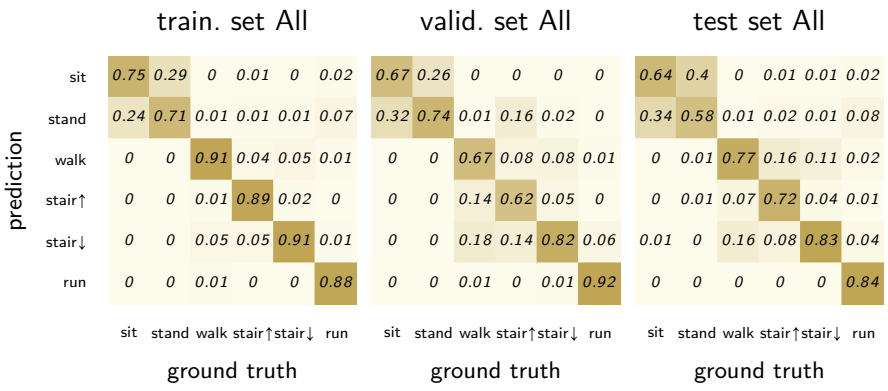


Figure 4.21. Heatmaps for confusion matrices separated by training set, validation set, and test set with aggregated repositories.

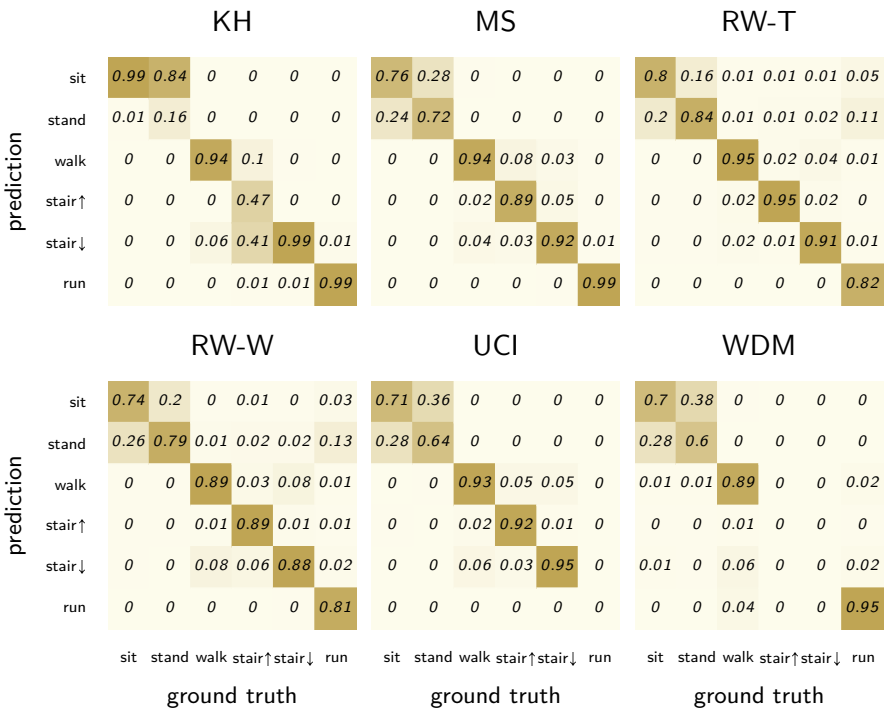


Figure 4.22. Confusion matrices (heatmaps) for the training set, by source.

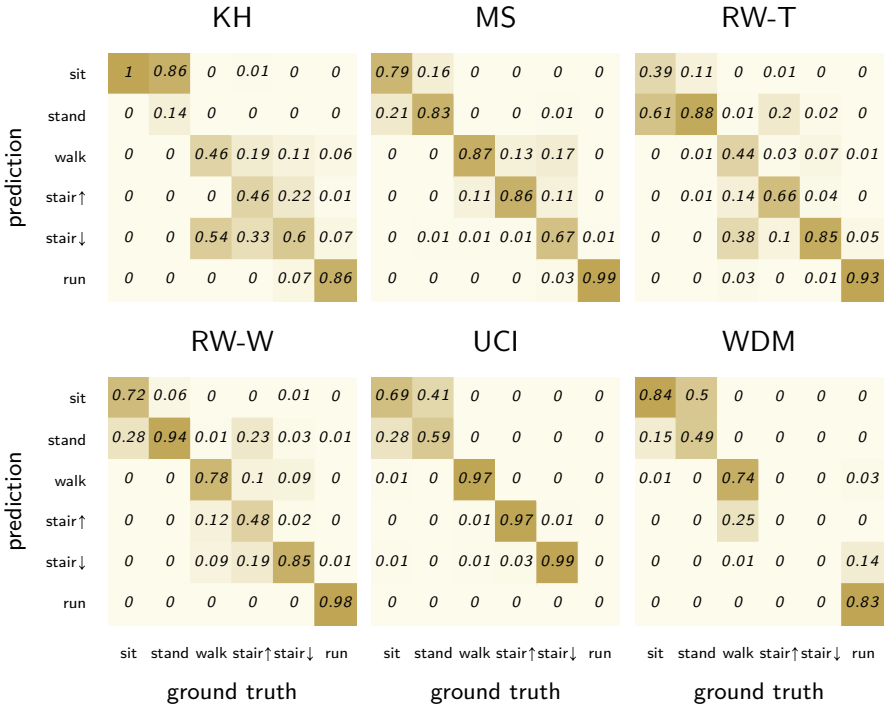


Figure 4.23. Confusion matrices (heatmaps) for the validation set, by source.

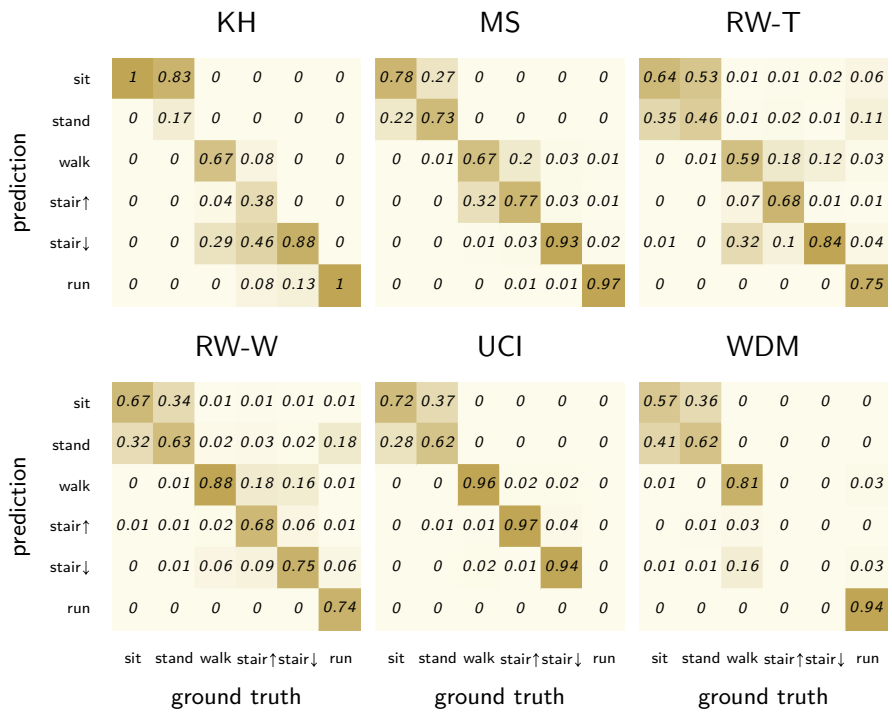


Figure 4.24. Confusion matrices (heatmaps) for the test set, by source.

4.2 FEDERATED APPROACH

The experiments were conducted by progressively moving away from the centralized model, organizing the training in 20 rounds, with 10 training epochs per round (§ 4.2.1).[3]

Given the high computational times required across the different distinct real users of DAGHAR (simulations are carried out on a single PC, an not on parallelizable separated devices), and assuming a large number of rounds needed to achieve sufficient training performance, in § 4.2.2 the federated approach was tested just on one of the single datasets available in DAGHAR. The RW-T dataset was chosen for its high number of uniform examples.[4]

---

distributions and seeks to preserve these relationships when projecting the data into a lower-dimensional space, where affinities are represented by Student's t-distributions. As a result, natural clusters emerge because neighborhood relationships are maintained: the probability that a point is close to another in the original high-dimensional space is computed using a Gaussian distribution, and t-SNE optimizes the embedding to approximate these probabilities using a Student-t distribution. Compared to a Gaussian distribution, the Student-t distribution enables better point separation in the low-dimensional space due to its heavier tails.

3 From the overall DAGHAR training set, all users who did not perform at least 4 of the 6 considered types of actions, and with fewer than 2 examples per category, were removed. This reduced the dataset to 109 distinct users.

4 Actually, among the DAGHAR subsets, RW-W (with smartphones positioned at the waist) contains the largest number of examples. However, RW-T (RealWorld-Tight) can offers a more realistic scenario, as the thigh position simulate a smartphone being carried in a pants pocket.

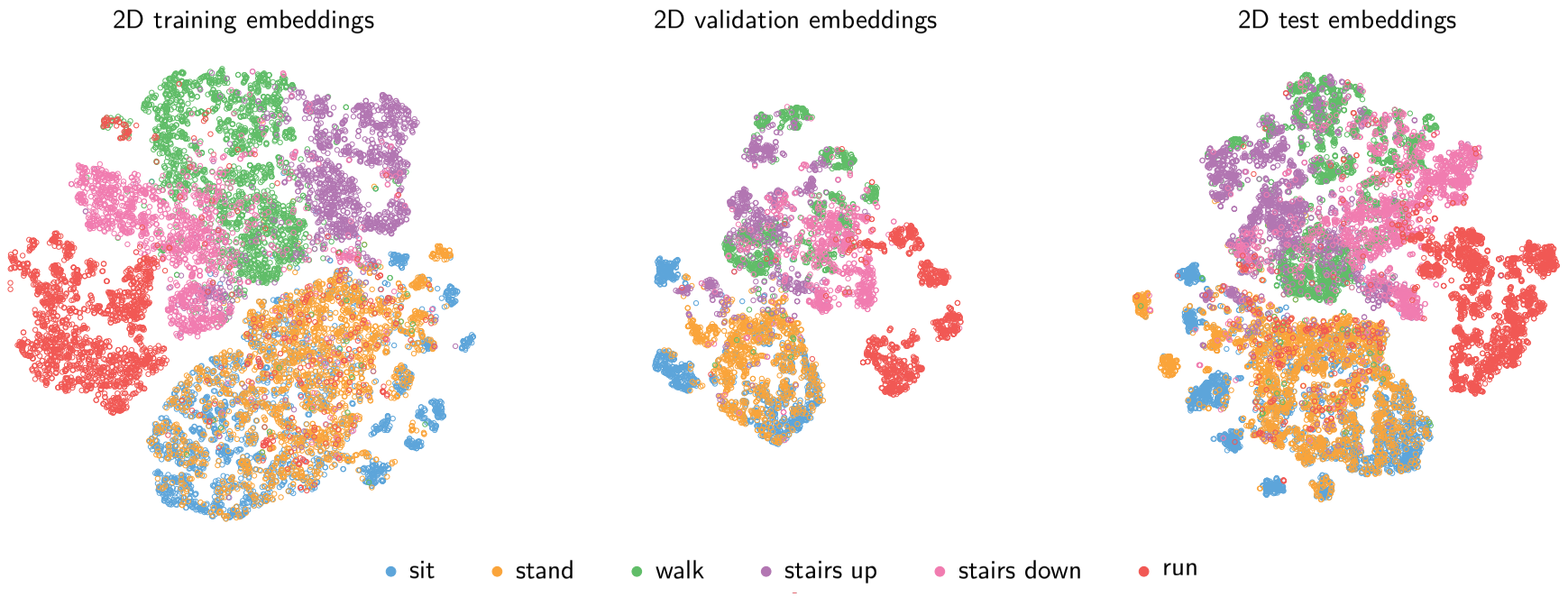


Figure 4.25. 2D projection via t-SNE of the descriptors consisting of the LSTM last internal state.

Across all experiments, the adopted model mirrors the best centralized architecture: a bidirectional LSTM with two hidden layers (64 units each) and an MLP classifier with 128 neurons, with the centralized learning rate applied to all clients.

4.2.1 *Federated Learning with Uniform Clients*

The samples from the complete DAGHAR training set were uniformly distributed across K users, where K ranged from 1 to 20.[5] This distribution was stratified solely based on the performed activities. We then monitored the overall training progression from the server's perspective, evaluating the loss and accuracy on the global DAGHAR validation and test sets, exploiting all the clients in the training phase.

Figs. 4.26–4.27 illustrate both the validation set loss as a function of training rounds, considering the different numbers of uniform clients (K), and the corresponding test set accuracy. Notably, the accuracy values presented in these figures correspond to the round with minimum loss observed on the server validation set: A decreasing trend in accuracy is observed as the number of clients increases, suggesting that achieving the highest accuracy scores (attained with fewer clients) would likely require additional training rounds.

4.2.2 *Federated Learning with RW-T Dataset*

Several experiments were conducted, varying the percentage of clients used for training in each round (40%, 50%, and 60%) and the number of epochs per round (5, 10, 20). Tab. 4.5 summarizes the results:[6]

- "epochs" reports the number of epochs per round;
- "clients %" shows the percentage of clients trained per round;
- "round" refers to the round where accuracy values were recorded;
- "valid." and "test" indicate, respectively, the accuracy on the server's validation and test sets, while "avg" displays their weighted average based on the examples in the two sets;
- "aggr. eval." reports the server-averaged accuracy calculated from individual client evaluations (`evaluate()` in Flower), weighted by the number of examples per user.

The values correspond to the round with the minimum weighted average loss across the validation set (2 users) and test set (3 users). The small validation set introduces noise, requiring smoothing via

5 The case of a single user is tested just to have an idea of the best achievable result.
6 In all these experiments the batch size was lowered from 256 to 64, due to the smaller number of examples in each real user.

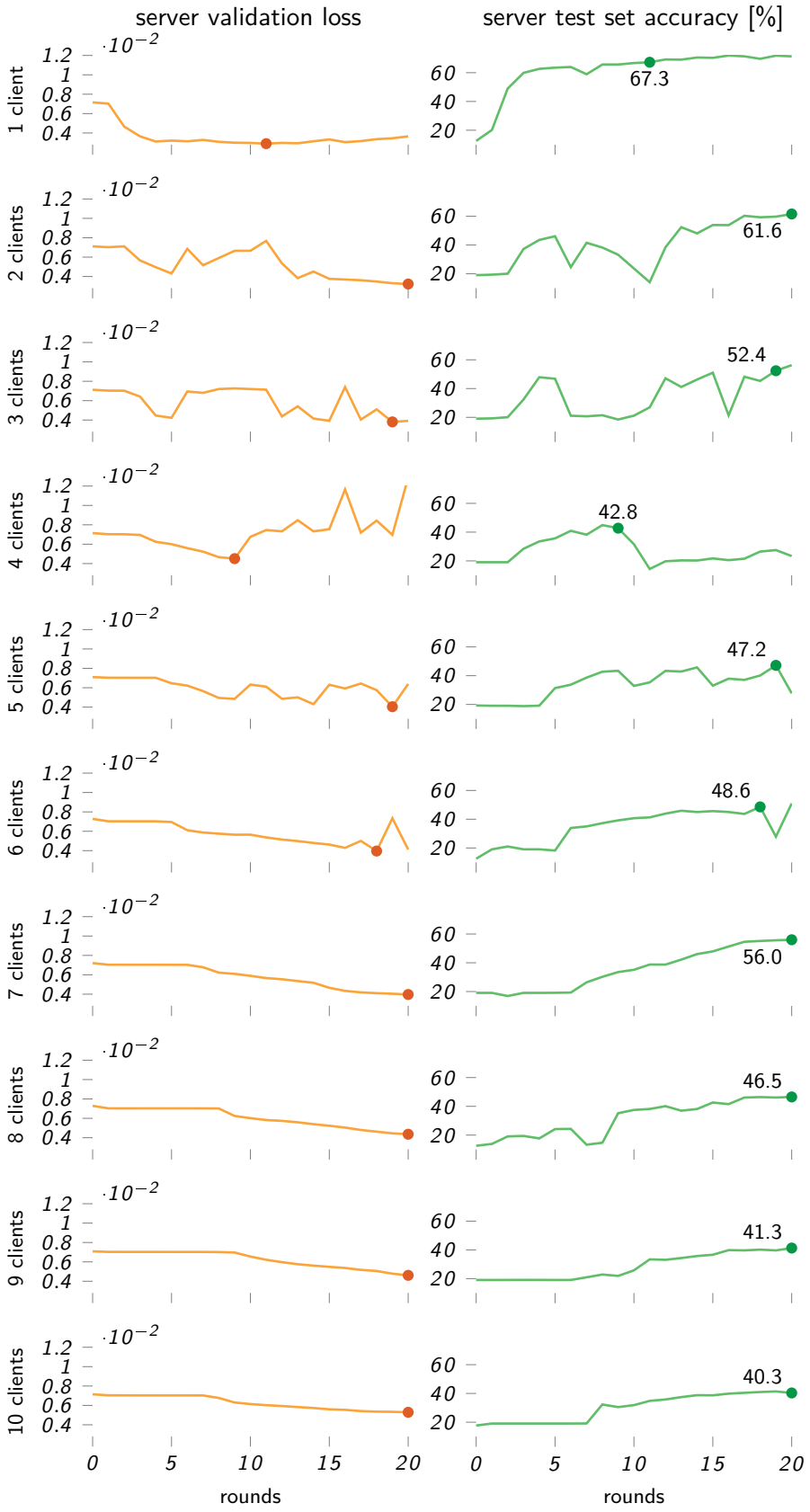


Figure 4.26. Validation loss and accuracy on test set evaluated at server updates, ranging from 1 to 10 uniform users.

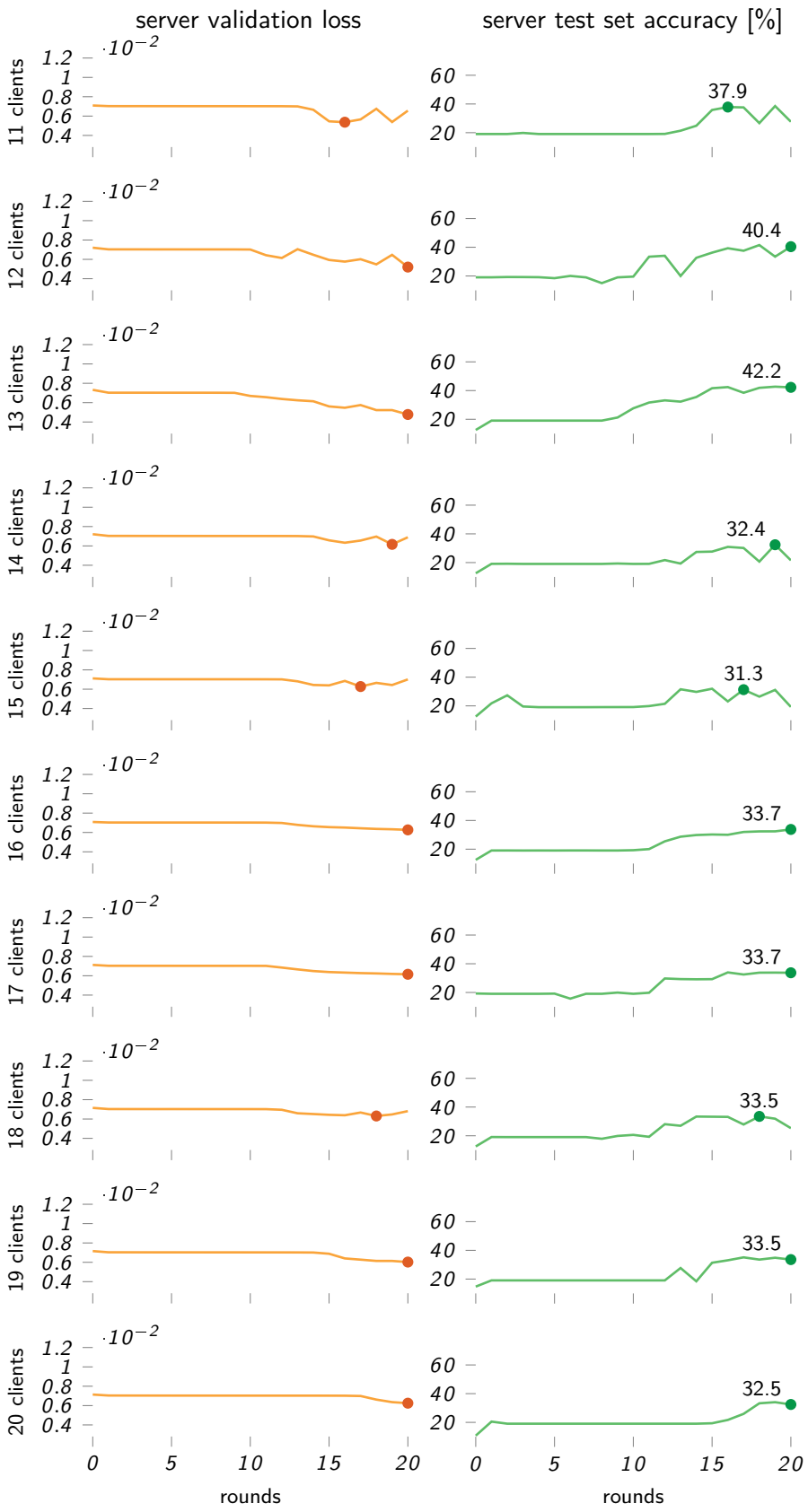


Figure 4.27. Validation loss and accuracy on test set evaluated at server updates, ranging from 11 to 20 uniform users.

Table 4.5. Accuracies on RW-T dataset evaluated for several combinations of epochs per round and percentage of clients taking part to fit phase.

			accuracy %				
epochs	clients %	round	valid.	test	avg	aggr. eval.	details
5	40	98	25.2	67.4	50.6	51.4	Fig. 4.28
	50	95	32.4	63.0	50.7	57.7	Fig. 4.29
	60	71	40.1	54.5	48.7	45.9	Fig. 4.30
10	40	52	42.9	65.6	56.6	60.4	Fig. 4.31
	50	81	51.7	64.3	59.3	63.0	Fig. 4.32
	60	49	36.1	57.8	49.1	57.1	Fig. 4.33
20	40	77	65.5	63.4	64.3	62.3	Fig. 4.34
	50	50	50.4	71.6	63.1	**68.8**	Fig. 4.35
	60	56	52.9	64.3	59.7	66.6	Fig. 4.36

weighted averages of both losses; likewise, accuracy values were averaged between the server's validation and test results.

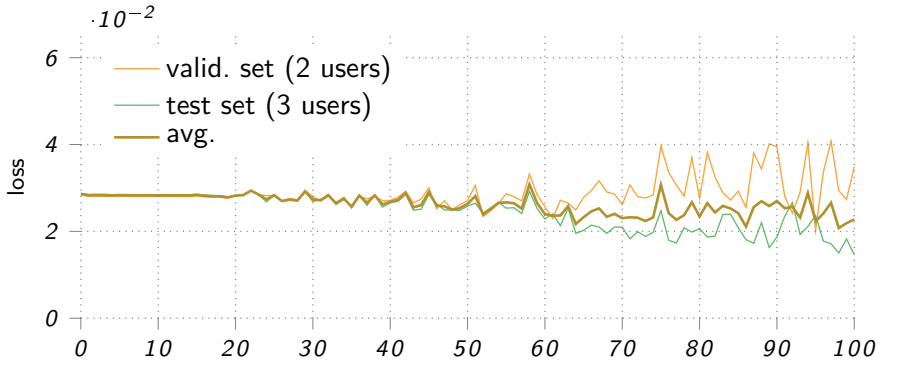
Figs. 4.28-4.36 detail the metrics computed per round, including aggregated loss and accuracy during the fit phase. From the fit metrics we can argue that, as the number of epoch per round increases, overfit occur earlier, while decreases as the number of clients partecipating in the fit process increases.

While RW-T contains a large number of examples per client, it has the drawback of having few users overall (in addition to the 2 in the validation set and 3 in the test set, there are 10 users in the training set). However, having reserved 30% of each of the 10 training users' examples as their local validation set, the aggregated accuracy on these local validation sets can serve as a surrogate for the values measured on the server's reserved validation and test sets.

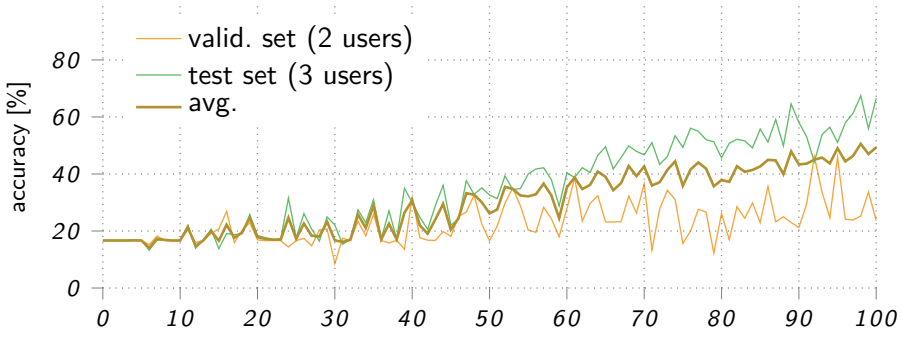
Tab. 4.5 shows that the combination of 20 epochs per round using 50% of the clients achieves the highest accuracy measured on the aggregated values obtained during the evaluate phase across all clients, at round 50. Let's now analyze the client-by-client metrics in more detail for this specific combination of epochs per round (20) and clients per fit (50%). Tab. 4.6 reports the accuracy evaluated by every client at round 50 on their validation set: The results are quite good, indeed with just one really bad score (42.8%); the details of the each client metrics are finally shown in Fig. 4.37.

Table 4.6. Accuray value evaluated by each client at round 50.

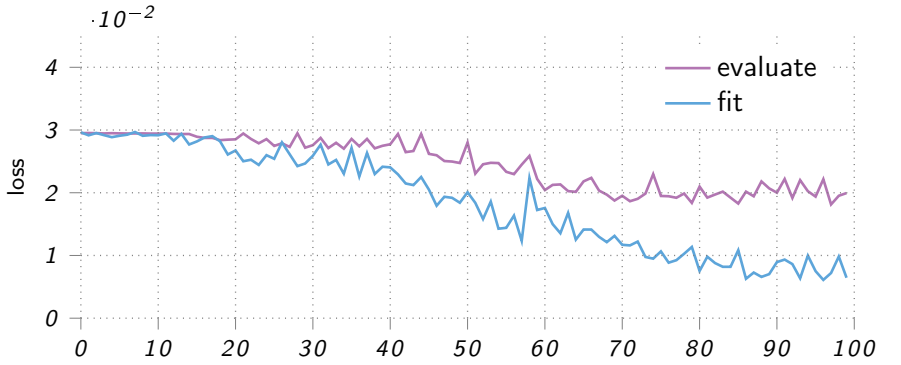
	users									
	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
accuracy %	71.4	42.8	55.7	52.9	70.1	75.1	80.7	85.7	76.5	74.4



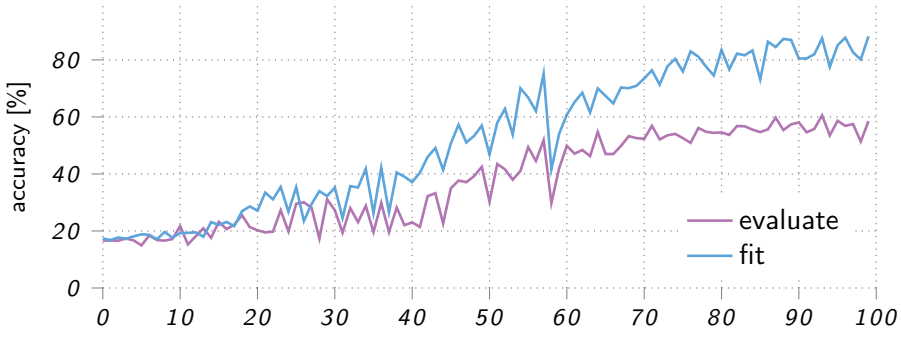
(a) server loss



(b) server accuracy

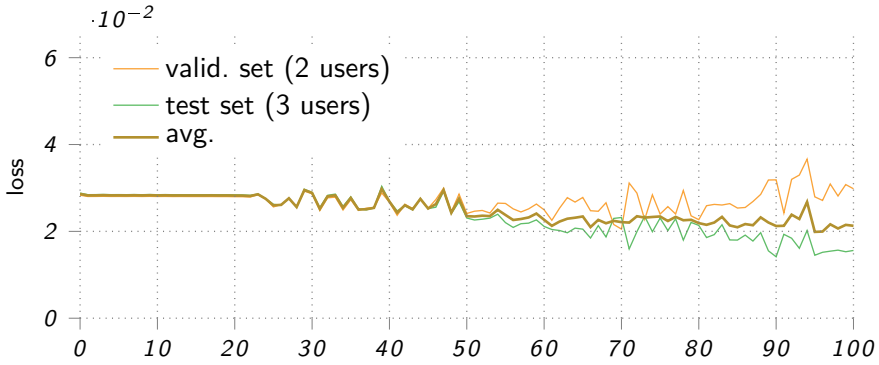


(c) aggregated loss

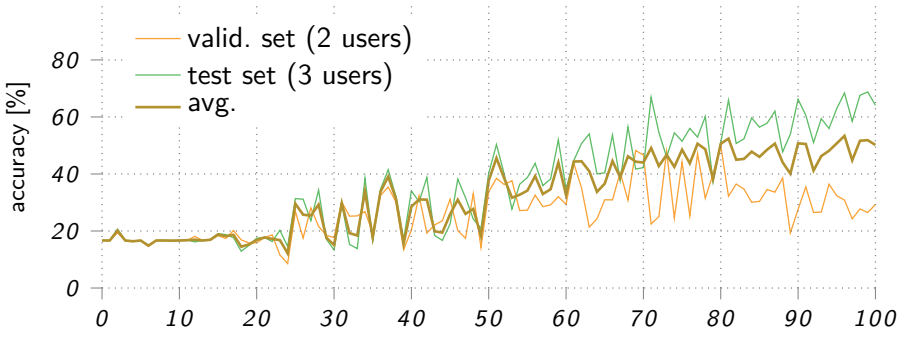


(d) aggregated accuracy

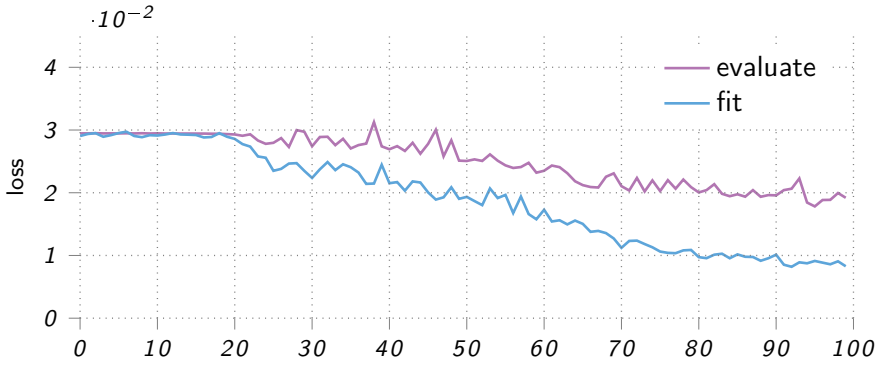
Figure 4.28. Validation loss and accuracy on test set evaluated at server updates on 4 clients, for RW-T dataset (5 epochs per round).



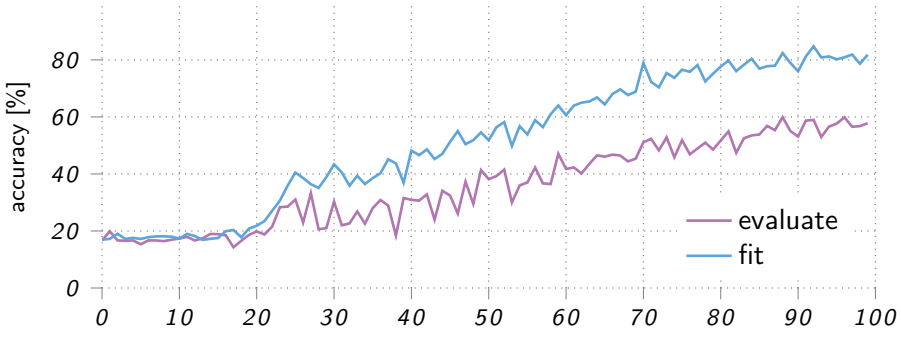
(a) server loss



(b) server accuracy

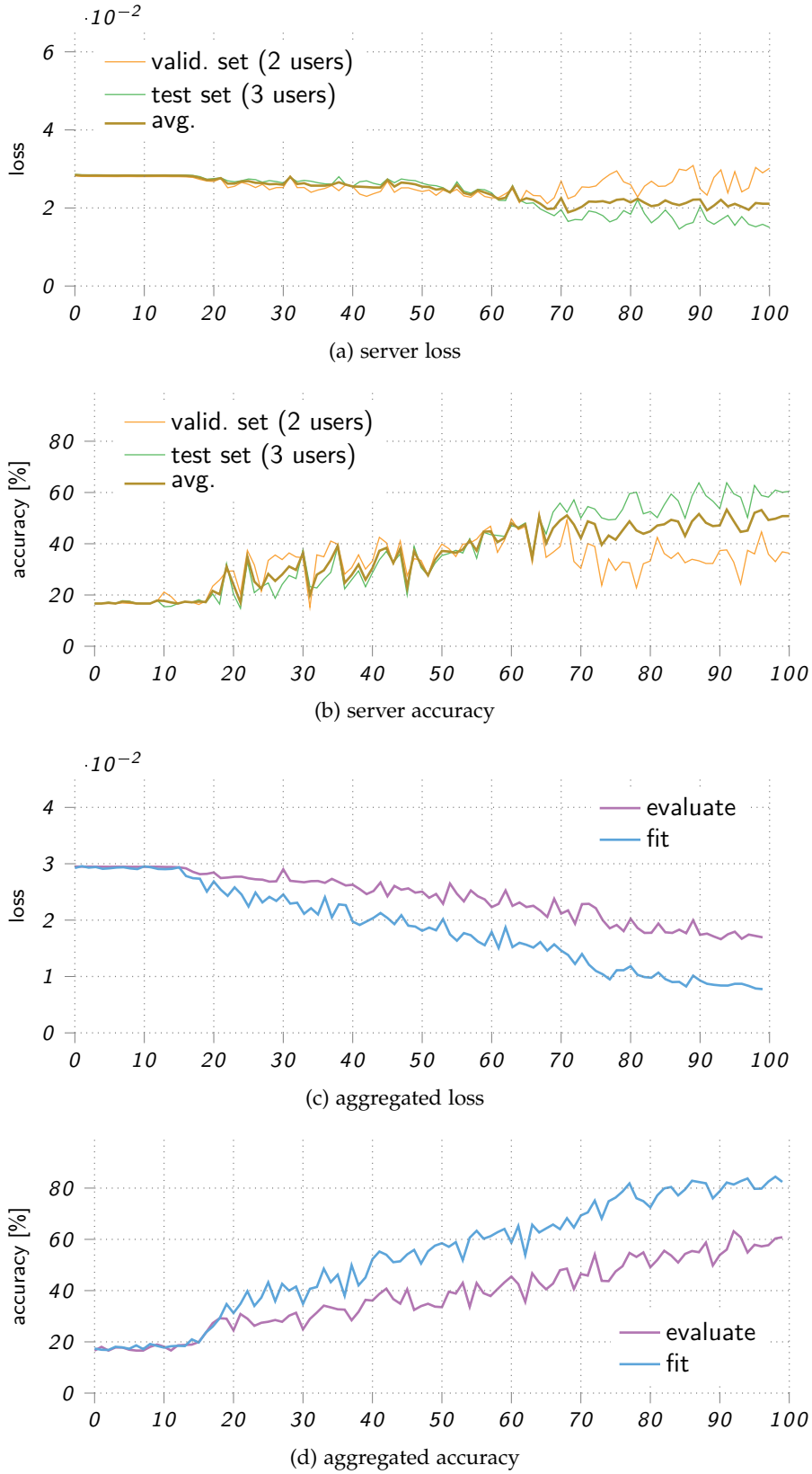


(c) aggregated loss



(d) aggregated accuracy

Figure 4.29. Validation loss and accuracy on test set evaluated at server updates on 5 clients, for RW-T dataset (5 epochs per round).



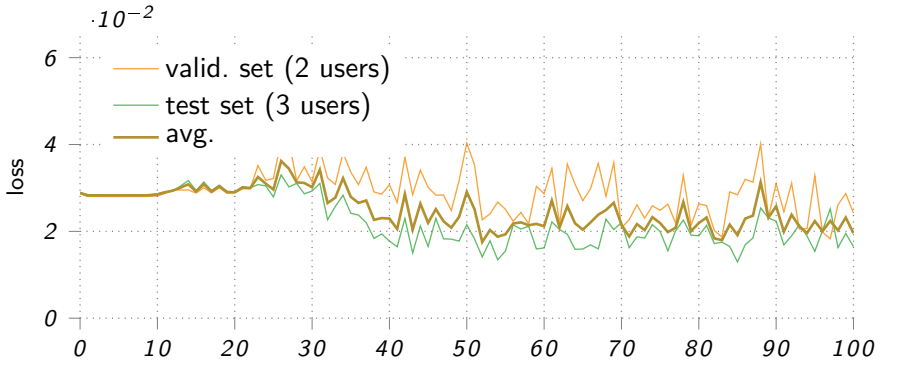
(a) server loss

(b) server accuracy

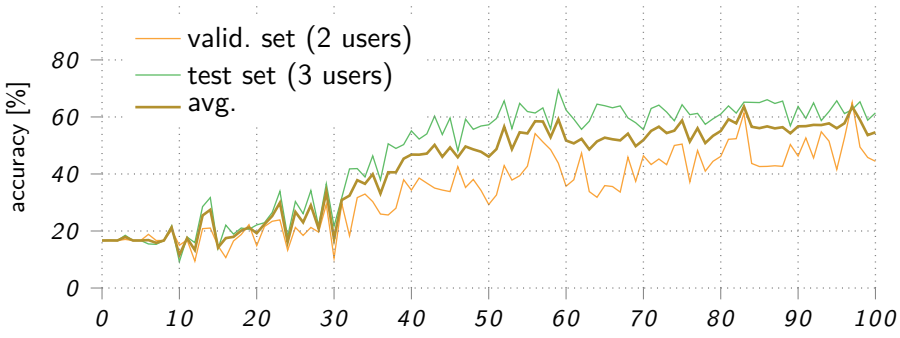
(c) aggregated loss

(d) aggregated accuracy

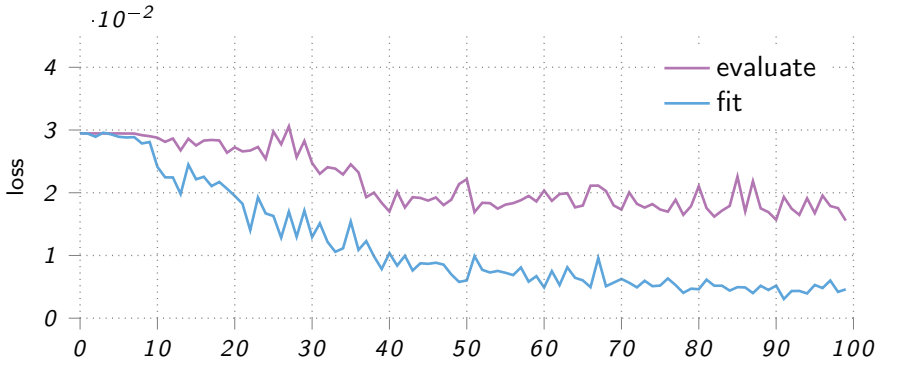
Figure 4.30. Validation loss and accuracy on test set evaluated at server updates on 6 clients, for RW-T dataset (5 epochs per round).



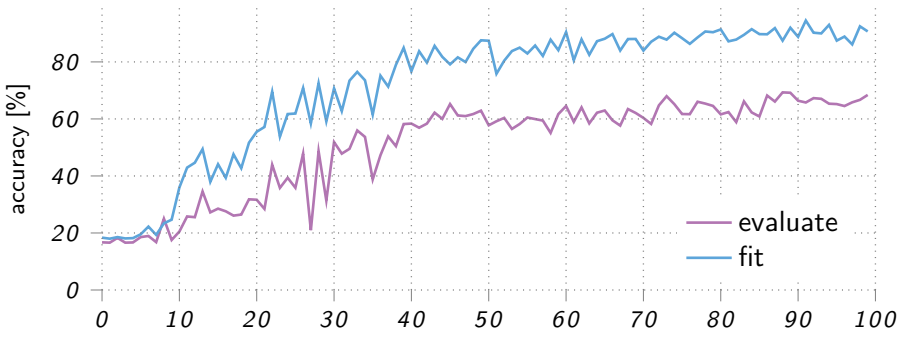
(a) server loss



(b) server accuracy

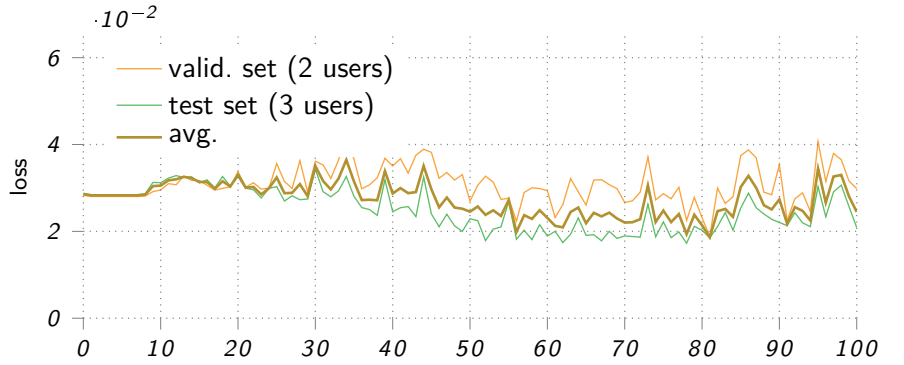


(c) aggregated loss

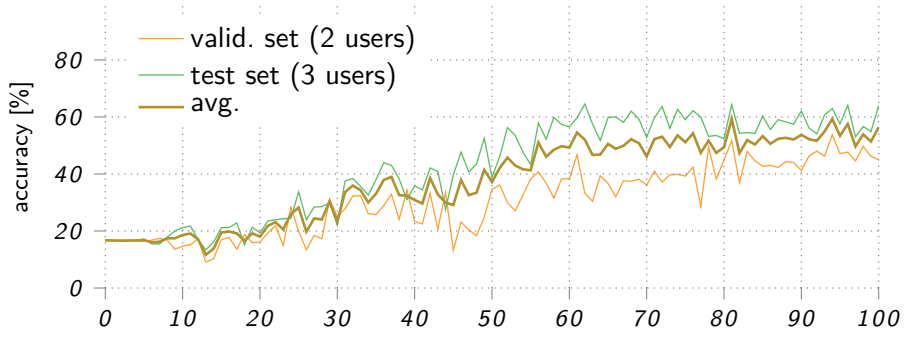


(d) aggregated accuracy

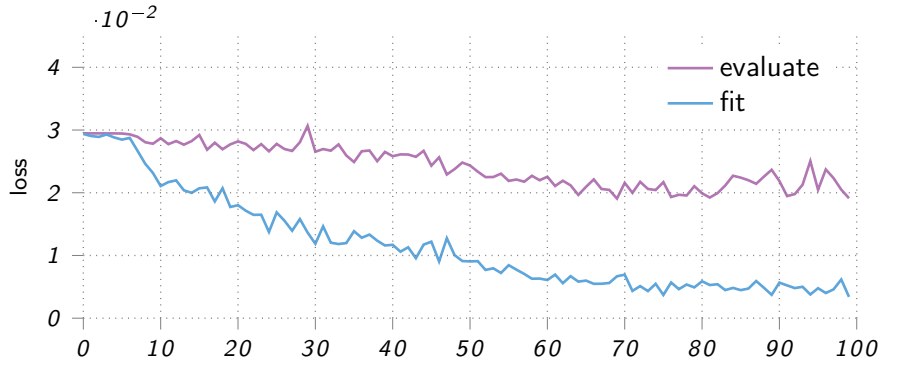
Figure 4.31. Validation loss and accuracy on test set evaluated at server updates on 4 clients, for RW-T dataset (10 epochs per round).



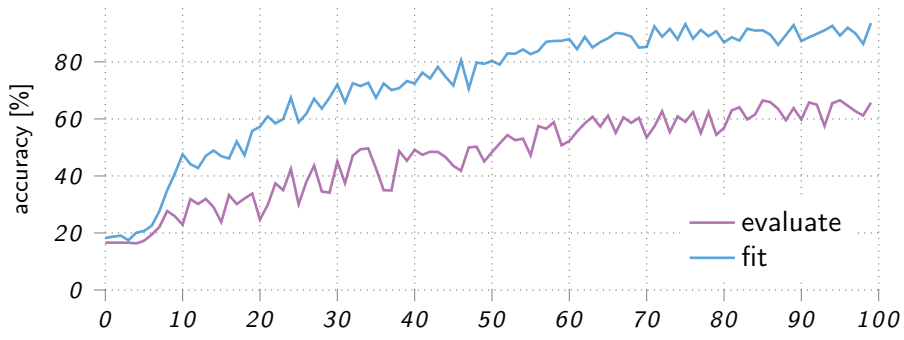
(a) server loss



(b) server accuracy

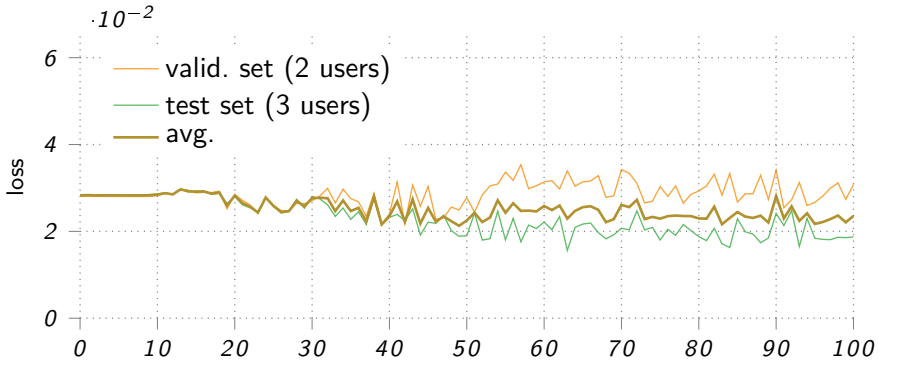


(c) aggregated loss

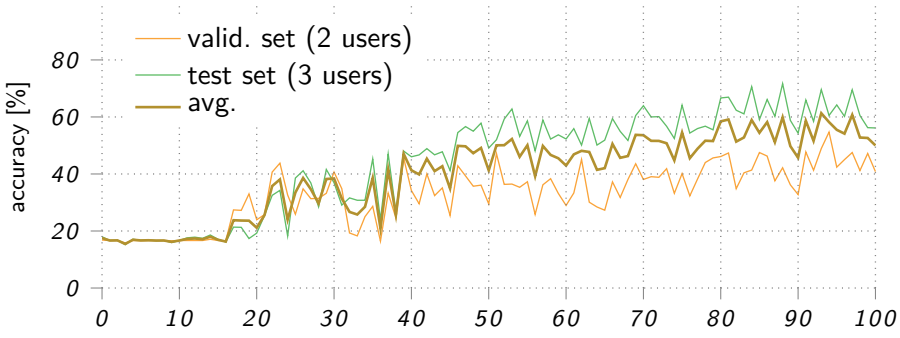


(d) aggregated accuracy

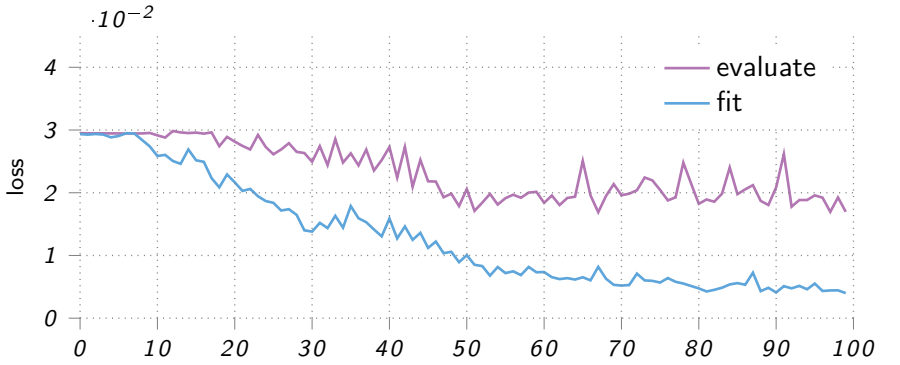
Figure 4.32. Validation loss and accuracy on test set evaluated at server updates on 5 clients, for RW-T dataset (10 epochs per round).



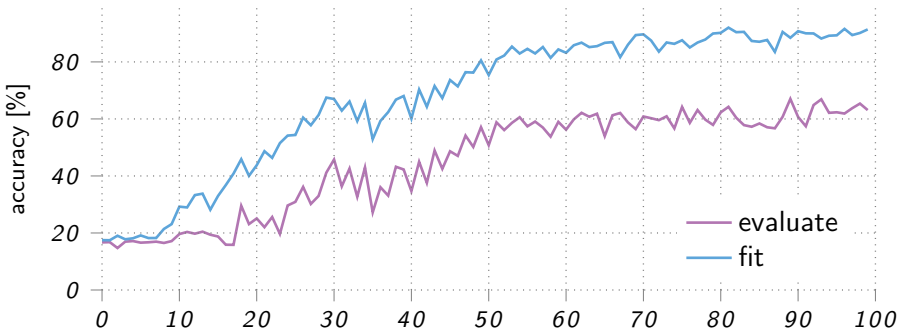
(a) server loss



(b) server accuracy

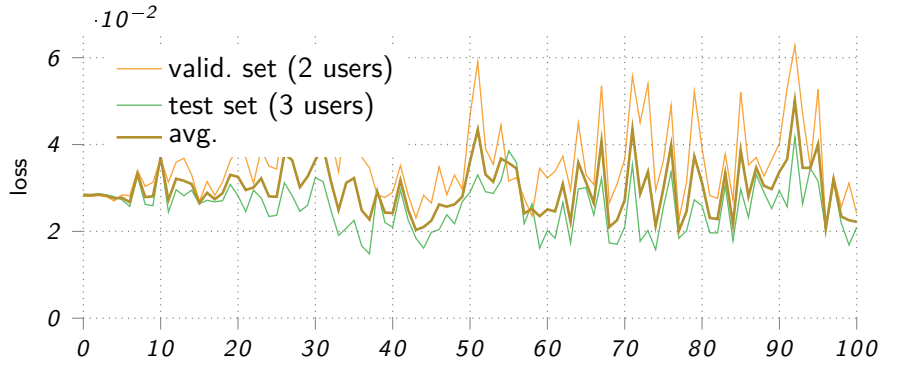


(c) aggregated loss

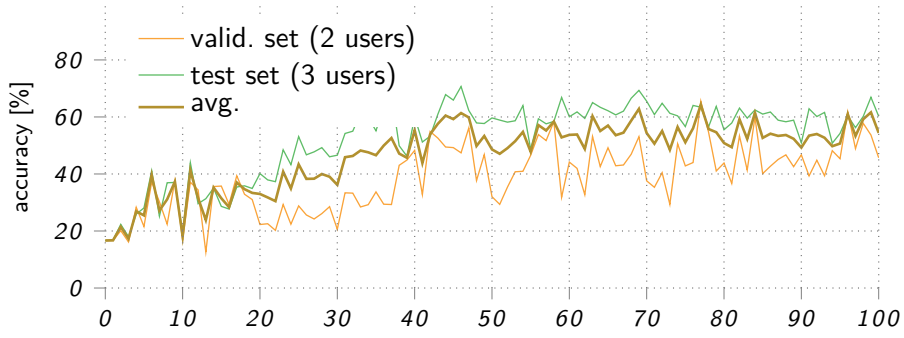


(d) aggregated accuracy

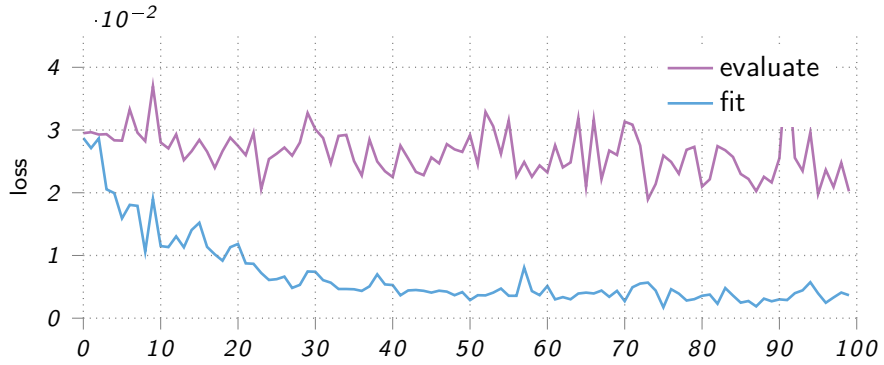
Figure 4.33. Validation loss and accuracy on test set evaluated at server updates on 6 clients, for RW-T dataset (10 epochs per round).



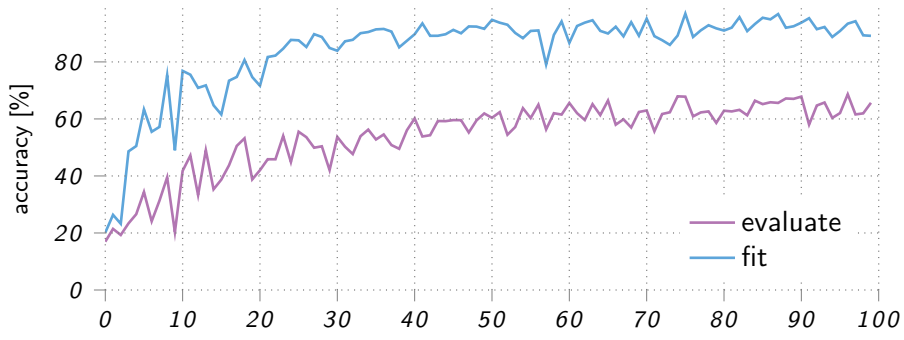
(a) server loss



(b) server accuracy

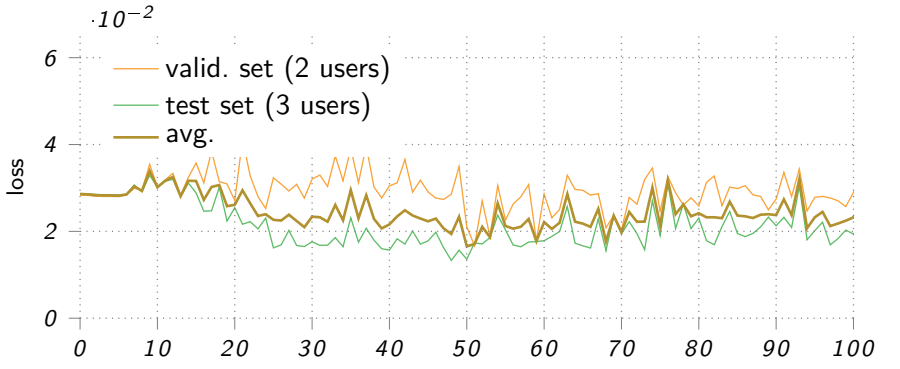


(c) aggregated loss

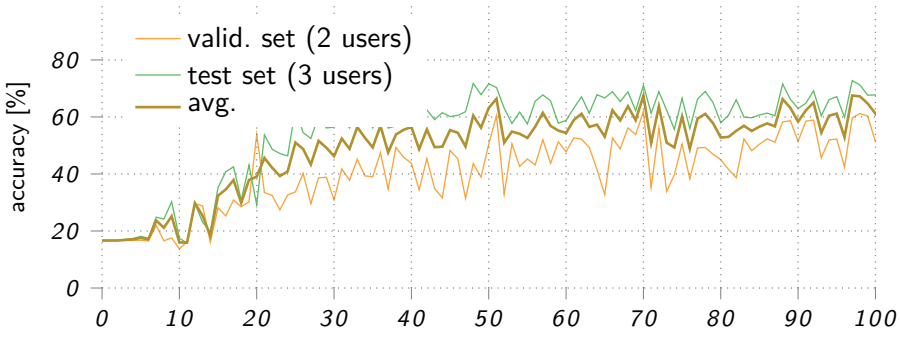


(d) aggregated accuracy

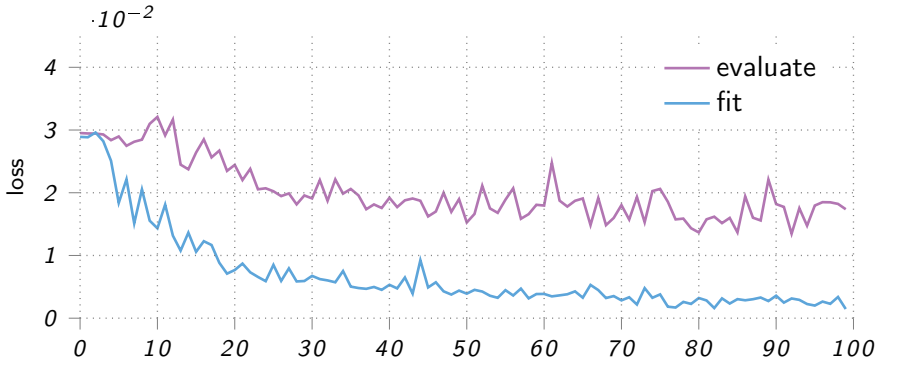
Figure 4.34. Validation loss and accuracy on test set evaluated at server updates on 4 clients, for RW-T dataset (20 epochs per round).



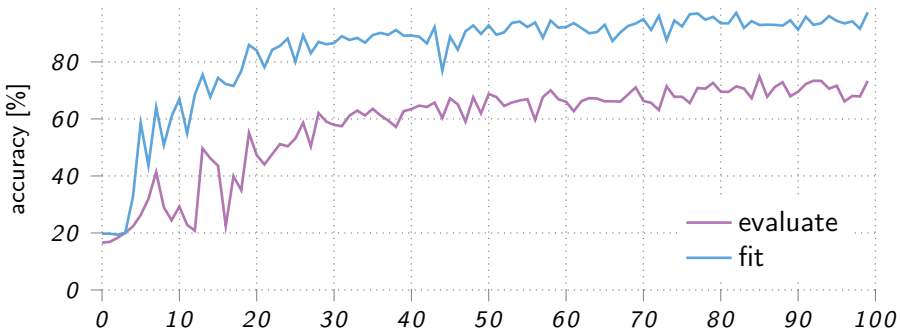
(a) server loss



(b) server accuracy

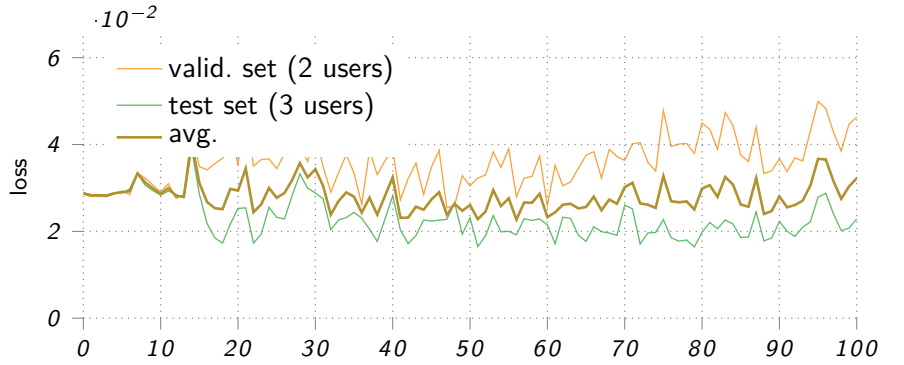


(c) aggregated loss

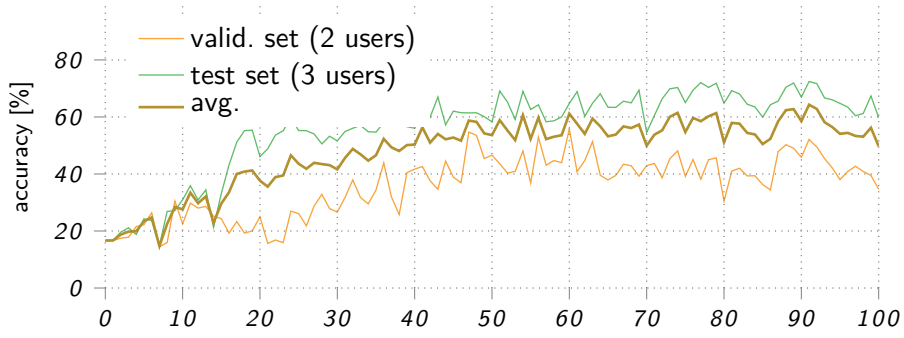


(d) aggregated accuracy

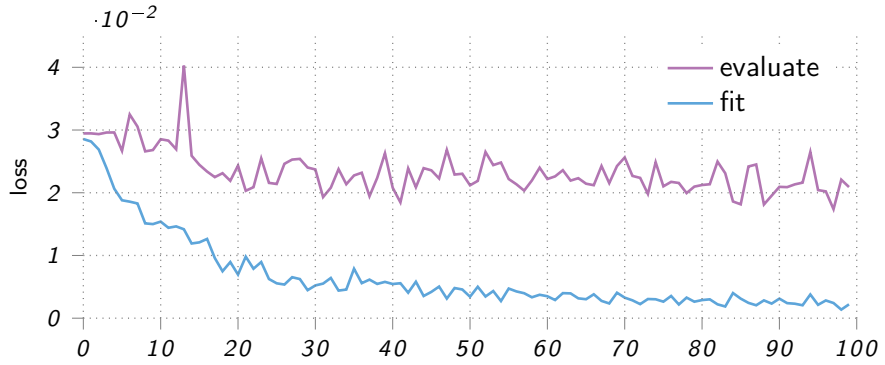
Figure 4.35. Validation loss and accuracy on test set evaluated at server updates on 5 clients, for RW-T dataset (20 epochs per round).



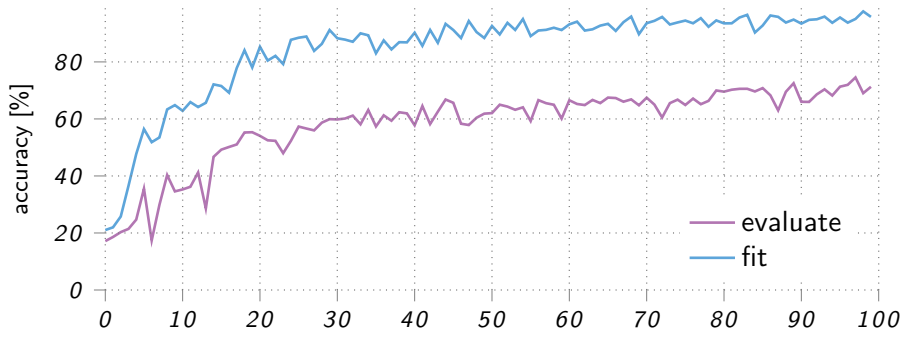
(a) server loss



(b) server accuracy



(c) aggregated loss



(d) aggregated accuracy

Figure 4.36. Validation loss and accuracy on test set evaluated at server updates on 6 clients, for RW-T dataset (20 epochs per round).

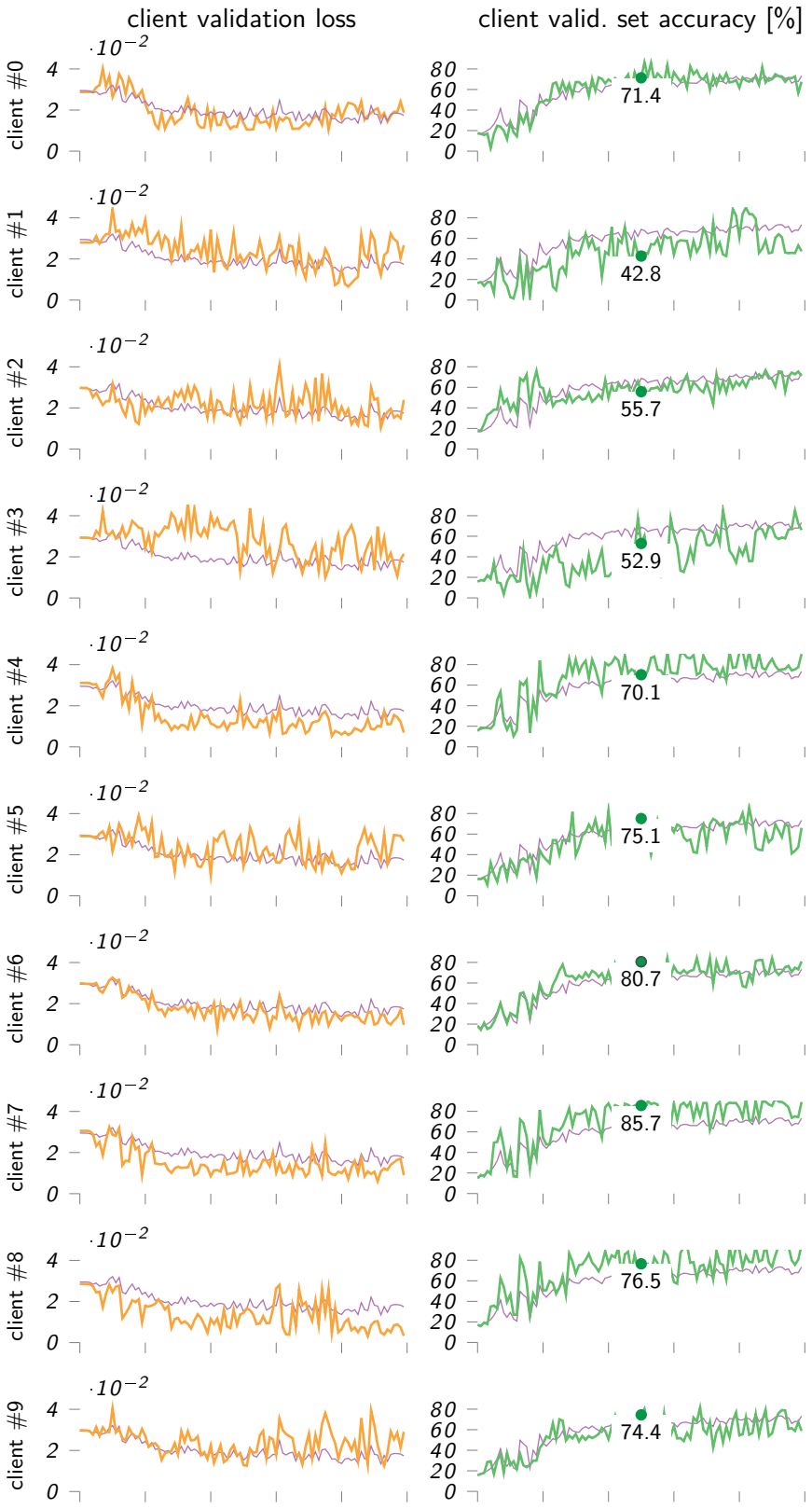


Figure 4.37. Validation loss and accuracy on validation set evaluated by each client user. In purple, respectively in (a) and (b) the aggregated loss and accuracy.

5 CONCLUSIONS

This work demonstrated the effective applicability of Federated Learning (FL) to sensor-based Human Activity Recognition (HAR): Both centralized and federated approaches were explored, using a deep learning architecture based on a bidirectional LSTM and an MLP.

The results obtained on benchmark datasets such as RealWorld (RW-T) let us hope that FL represents a valid solution for addressing the challenges related to privacy and data decentralization in HAR.

5.1 FUTURE WORK

Although this work has found an effective applicability of the federated method in HAR contexts, there are numerous aspects to improve or investigate further:

- First of all, trying to increase the ability to discriminate between the "sit" and "stand" actions,
  - Perhaps introducing data augmentation mechanisms, as described in § 5.1.1, to prevent early overfitting;
  - Maybe by adding a convolutional network before the recurrent model to obtain denser and richer embeddings;
  - Employing a Transformer instead of the LSTM model;
- Furthermore, Self-Supervised Learning techniques, (such as Masked Autoencoders, which have proven effective with limited data), could be take into account in order to improve training on resource-constrained FL clients;
- Finally, the federated approach should be tested (taking a sufficiently long processing time) on all users made available by DAGHAR (not just RW-T).

Furthermore, as an alternative to the standard approach that employs recurrent models, a different method could exploit 2D convolutional networks applied to spectrograms calculated on the signals related to each action, as described in more detail in § 5.1.2.

5.1.1 *Data Augmentation*

To apply data augmentation to a HAR dataset, we could attempt to add white noise to the data as implemented in List. 5.1.

Listing 5.1: Generating random noise to add to the examples.

```
import numpy as np
from scipy.signal import butter, filtfilt

fs = 20 # Sampling freq. (Hz)
nyquist = fs / 2 # Nyquist Frequency
cutoff = 0.75*nyquist # Cut Freq.

# Low-pass filter
b, a = butter(4, cutoff / nyquist, btype='low')

# Noise generation
sigma = 0.01
duration = 3 # Duration in seconds
n_samples = int(fs * duration)
noise = np.random.normal(0, sigma, n_samples) # Gaussian (white)
    noise
filtered_noise = filtfilt(b, a, noise) # to be added
```

5.1.2 *Classification with Spectrograms*

Below are proposed some ideas concerning spectrograms with HAR.

Preprocessing

The 3-second slices from the DAGHAR benchmark are processed to construct spectrograms, one for each x-y-z axis of the acquired signals:

1. Windowing on each signal slice, composed of $N = 60$ samples, for example using a Hann window [18, p. 657] (or using windows that taper to zero at the edges)
$$w_n = \frac{1}{2}\left(1 - \cos\frac{2\pi n}{N}\right)$$
2. FFT applied on a moving window with a 1-second radius, with zero-padding of 1 second on both sides, centered on sample $n = 0 \ldots N - 1$
3. Each FFT constitutes a column of the spectrogram (to be evaluated whether to consider only the amplitude of the spectra or also the phase)
4. Normalize the spectrograms so that the most intense component across the entire dataset assumes a value of 1?

Applying the FFT over a 2-second span, as described in point 2, provides a resolution of 0.5 Hz: with data sampled at $s_r = 20$ Hz, Shannon's theorem limits spectral content to frequencies below $s_r/2$. Thus, each spectrum contains $10\,\text{Hz}/0.5\,\text{Hz} = 20$ samples, and the spectrograms for each channel measure 20×60 pixels.

Spectrogram Visualization

Besides representing each channel's spectrogram separately using appropriate colormaps, an interesting approach could be encoding the values of the three x-y-z components into two images—one for the accelerometer signal and one for the gyroscope—where the RGB channels correspond to the spectrograms of the three axial components.

5.1.3 *Other HAR Datasets*

In the initial phases of this work, several additional HAR datasets were identified. Hopefully these datasets can be converted to align with DAGHAR standards, thereby expanding the pool of users and enabling a more realistic and comprehensive scenario. Below is a list of these datasets, categorized by the sites where they are accessible.

UNIMI

- https://everywarelab.di.unimi.it/index.php/research/datasets/228-domino-a-dataset-for-context-aware-human-activity-recognition-using-mobile-devices

KAGGLE

- https://www.kaggle.com/datasets/chumajin/heterogeneity-human-activity-recognition-dataset?select=Phones_accelerometer.csv
- https://www.kaggle.com/datasets/nickpisarevds/smartphonebased-recognition-of-hapt
- https://www.kaggle.com/datasets/impapan/human-activity-recognition-using-smartphone

IEEE

- https://ieee-dataport.org/documents/wild-shard-smartphone-sensor-based-human-activity-recognition-dataset-wild
- https://ieee-dataport.org/open-access/harsense-statistical-human-activity-recognition-dataset

BIBLIOGRAPHY

[1] Flower A Friendly Federated AI Framework. https://flower.ai/.

[2] G. Ausiello, A. Marchetti Spaccamela, and M. Protasi. *Teoria e progetto di algoritmi fondamentali*. Franco Angeli, 1996.

[3] Liming Chen, Jesse Hoey, Chris D. Nugent, Diane J. Cook, and Zhiwen Yu. Sensor-based activity recognition. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):790–808, 2012. https://ieeexplore.ieee.org/document/6208895.

[4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. Draft, 3rd edition, 2025. Online manuscript released January 12, 2025 https://web.stanford.edu/~jurafsky/slp3/.

[6] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. Activity recognition using cell phone accelerometers. *ACM SIGKDD Explorations Newsletter*, 12(2):74–82, 2011. https://dl.acm.org/doi/10.1145/1964897.1964918.

[7] WIreless Sensor Data Mining Lab. Wisdm: Wireless sensor data mining dataset. https://www.cis.fordham.edu/wisdm/dataset.php, 2012.

[8] Mohammad Malekzadeh. Motionsense dataset. https://github.com/mmalekzadeh/motion-sense, 2019.

[9] Mohammad Malekzadeh, Richard G. Clegg, Andrea Cavallaro, and Hamed Haddadi. Mobile sensor data anonymization. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, page 49–58, New York, NY, USA, 2019. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3302505.3310068.

[10] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Aarti Singh and Xiaojin (Jerry) Zhu, editors, *Proceedings of the 20th International*

Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA, volume 54 of *Proceedings of Machine Learning Research*. PMLR, 2017. http://proceedings.mlr.press/v54/mcmahan17a.html.

[11] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., 1997. https://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/mlbook.html.

[12] Abdullah-Al Nahid, Niloy Sikder, and Ibrahim Rafi. Ku-har: An open dataset for human activity recognition. https://data.mendeley.com/datasets/45f952y38r/5, 2021.

[13] Otávio Napoli, Dami Duarte, Patrick Alves, Darlinne Hubert Palo Soto, Henrique Evangelista de Oliveira, Anderson Rocha, Levy Boccato, and Edson Borin. A benchmark for domain adaptation and generalization in smartphone-based human activity recognition. *Scientific data*, 11(1):1192–1207, 2024. https://www.nature.com/articles/s41597-024-03951-4.

[14] Otávio Napoli, Dami Duarte, Patrick Alves, Darlinne Hubert Palo Soto, Henrique Evangelista de Oliveira, Anderson Rocha, Levy Boccato, and Edson Borin. Daghar: A benchmark for domain adaptation and generalization in smartphone-based human activity recognition. https://zenodo.org/records/13987073, 2024.

[15] University of Mannheim. Realworld (har). https://www.uni-mannheim.de/dws/research/projects/activity-recognition/dataset/dataset-realworld/, 2016.

[16] Ian Pointer. *Programming PyTorch for Deep Learning*. O'Reilly Media, Inc., 2019.

[17] Ek S. Presotto, R. Civitarese, G. Portet, F. Lalanda, and P. Bettini. Comparing self-supervised learning techniques for wearable human activity recognition. *CCF Transactions on Pervasive Computing and Interaction*, 2025. https://link.springer.com/article/10.1007/s42486-024-00182-9.

[18] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, USA, 3 edition, 2007. https://numerical.recipes/book.html.

[19] J. Reyes, L. Oneto, A. Sama, X. Parra, and D. Anguita. Transition-aware human activity recognition using smartphones. *Neurocomputing*, 171:754–754, 01 2016. http://www.sciencedirect.com/science/article/pii/S0925231215010930.

[20] Jorge Reyes-Ortiz, Davide Anguita, Alessandro Ghio, Luca Oneto, and Xavier Parra. Human Activity Recognition Using Smartphones. UCI Machine Learning Repository, 2013. https://archive.ics.uci.edu/dataset/240/human+activity+recognition+using+smartphones.

[21] Leslie N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, 2017.

[22] Timo Sztyler and Heiner Stuckenschmidt. On-body localization of wearable devices: An investigation of position-aware activity recognition. In *2016 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–9. IEEE Computer Society, 2016. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7456521.

[23] Emanuele Trucco and Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall PTR, USA, 1998.

[24] Yonatan Vaizman, Katherine Ellis, and Gert Lanckriet. The ExtraSensory Dataset, 2016. http://extrasensory.ucsd.edu/.

[25] Yonatan Vaizman, Katherine Ellis, and Gert Lanckriet. Recognizing detailed human context in the wild from smartphones and smartwatches. *IEEE Pervasive Computing*, 16(4):62–74, 2017. https://ieeexplore.ieee.org/document/8090454.

[26] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008. https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf.

[27] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Lisha Hu. Deep learning for sensor-based activity recognition: A survey. *Pattern Recognition Letters*, 119:3–11, 2019. https://www.sciencedirect.com/science/article/pii/S016786551830045X.

[28] Gary Weiss, Kenichi Yoneda, and Thaier Hayajneh. Smartphone and smartwatch-based biometrics using activities of daily living. *IEEE Access*, PP:1–1, 09 2019. https://www.researchgate.net/publication/335785947_Smartphone_and_Smartwatch-Based_Biometrics_Using_Activities_of_Daily_Living.

[29] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. https://D2L.ai.