POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Computer Science

Intelligent Systems and Data Science

Data Science

Mateusz Darnowski
S18322

Enhancing Activity Recognition in m-Health Applications

Master's Thesis

Thesis Supervisor:
prof. dr hab. Grzegorz Marcin Wójcik

Warsaw, September 11, 2024

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Informatyka

Systemy Inteligentne i Data Science

Data Science

Mateusz Darnowski
S18322

Udoskonalanie Rozpoznawania Aktywności w Aplikacjach m-Health

Praca magisterska

Promotor:
prof. dr hab. Grzegorz Marcin Wójcik

Warszawa, 2 września, 2024

Abstract

This work uses the MHEALTH dataset to improve activity recognition models through targeted hyperparameter tuning and machine learning techniques. The dataset includes motion and vital signs data from ten subjects performing twelve activities, recorded by sensors on the chest, wrist, and ankle. This study explores diverse tuning strategies and reports their performances on the classification task embedded in the dataset. After reporting the tuning results, this study proposes a refined model for classifying human activities from segments of sensor recordings, whose generalization is demonstrated on unseen data.

Keywords: Human Activity Recognition, HAR, MHEALTH Dataset, Machine Learning, Hyperparameter Tuning, Multi-class Classification

Abstract

Praca wykorzystuje zbiór danych MHEALTH do usprawnienia modeli rozpoznawania aktywności poprzez hiperparametryzację i techniki uczenia maszynowego. Zbiór danych zawiera dane dziesięciu uczestników wykonujących dwanaście aktywności rejestrowanych przez czujniki umieszczone na klatce piersiowej, nadgarstku i kostce. Praca analizuje strategie hiperparametryzacji i przedstawia ich wyniki w zadaniu klasyfikacji osadzonym w zbiorze danych. Po przedstawieniu wyników hiperparametryzacji, zaproponowany jest udoskonalony model klasyfikacji aktywności na podstawie wycinków z nagrań sensorów, którego generalizacja jest potwierdzona na danych niewidzianych wcześniej przez model.

Słowa kluczowe: Human Activity Recognition, HAR, MHEALTH, Uczenie Maszynowe, Hiperparametryzacja, Klasyfikacja Wieloklasowa

Contents

1. Introduction

1.1. Background

Development and integration of wearable sensor technologies with machine learning algorithms have allowed substantial improvements in mobile health applications over the years [1]. Most of these apps help monitor health metrics on a continuous basis, capturing valuable data for both the patient and healthcare provider. Mobile health systems function like traditional health platforms but utilize mobile devices, such as smartphones, tablets. For example, teledermatology applications enable patients to send descriptions and photos of surgical wounds or skin issues to a specialist via a smartphone [2]. These apllications can be deployed in the actual daily environment of a patient, even when cellular reception is poor because locally stored data will be transmitted once better connectivity resumes. As a result, mobile health applications could be particularly useful in situations where continuous adherence is necessary. The significance of activity recognition in m-health is important because it successfully helps monitor physical activities that are essential for applications like chronic disease management, rehabilitation, and fitness tracking [3].

Activity recognition is the detection of certain physical activities, such as walking, running, or sitting, using data from sensors. Motion and physiological signals, wearable devices with accelerometers, gyroscopes, magnetometers, and electrophysiological recordings (e.g., ECG SmartWatch) are involved. These sensors produce time-series data corresponding to the dynamics of human movement and physiological states. The challenge lies in accurately differentiating these activities from the sensor data, as this type of data can be noisy, high-dimensional, and complex [4]. High-quality health monitoring and feedback can predict patient outcomes and provide better care, but only if the activity recognition is reliable.

Deep learning methodologies have emerged as a powerful tool to address the accuracy and reliability of activity recognition models [5]. Due to the complexity and variability of sensor data, traditional machine learning models might not perform well. On the other hand, deep learning-based models like CNNs (Convolutional Neural Networks) and RNNs (Recurrent Neural Networks) have exhibited breakthrough results in detecting relevant features and understand temporal patterns. CNNs, on the other hand, are useful for learning spatial patterns from sensor

signals (time-series data), and RNNs, especially Long Short-Term Memory networks, have been successful in capturing sequential information and long-term dependencies to detect activities where they occur over a temporal span.

This study employs the MHEALTH dataset [6] in building and analyzing these models. The dataset contains full-body motion and vital signs data from ten subjects performing twelve different physical activities, from simple actions like standing still and sitting down to more advanced activities like running or jumping. Data for each of the participants is captured with sensors situated at the chest, left ankle, and right wrist to record a wide range of movements and physiological responses. Such a setup helps to create models that are more generalizable; hence, they can scale across people and activities. Utilizing the MHEALTH dataset for activity recognition, deep learning models can be examined and improved in both the diversity of activities and the detail within a single repository. Various activities provide great diversity to the dataset, and many modalities (accelerometer, ECG, magnetometer data, etc.) show subjects' physical condition from multiple angles. This multi-modality is beneficial for creating models that can use various kinds of data to increase accuracy in classification. On the other hand, this complicates model development because the information from several sensor modalities needs to be fused properly by the same model.

Hyperparameters have a massive impact on how the model gets trained and performs. Optimal hyperparameters can help improve the accuracy of the model, reduce overfitting, and enhance generalization capabilities. Extensive studies have shown that model performance can be largely influenced by fine-tuning of hyperparameters [7]. This research work sheds light on improving deep learning models for better classification accuracy and consistency of activity recognition systems through the adoption of this method. For hyperparameter tuning, different optimization methods are explored, which generate hyperparameters systematically to easily find the best configurations for those parameters.

1.2. Motivation

The study is motivated by the need to improve activity recognition models. Traditional machine learning models are not very adept at handling the complexity of such data, whereas deep learning techniques, particularly Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), have achieved significantly higher levels of accuracy and reliability. This work exploits the richness of the MHEALTH dataset by using its various sensor data to build models that effectively capture complex patterns. Much focus is on hyperparameter tuning since well-tuned parameters significantly improve model performance. Activity classification will be optimized, and in doing so, this work targets making the recognition of physical activities more accurate. This study aims to add valuable insights that could be applicable to similar datasets and contribute to the field of activity recognition.

1.3. Objectives

The studied dataset consists of recorded sessions of performed activities by different human subjects. While the dataset could be used for research on EKG tracking and m-health reasoning, this study focuses on the classification task. This work tests state-of-the-art approaches and algorithms of machine learning to find the best and most accurate model for identifying and classifying these activities. Additionally, this work focuses on the hyperparameter tuning process to find the best architecture and sampling methods. It tests the best ways to supervise model creation, determines the optimal lengths of input, and provides a discussion on the relevance of features and the segment sizes fed into the model. In pursuit of these objectives, the study will draw on a comprehensive literature review for identification and analysis of existent methods and approaches in human activity recognition; very recent advances in machine learning, deep learning, and sensor data processing techniques that concern physical activities classification.

This study will apply state-of-the-art machine learning algorithms and techniques to test and compare different activity classification models, including Convolutional Recurrent Neural Networks and Long Short-Term Memory networks. During this process, extensive hyperparameter tuning will be conducted. This includes adjusting the learning rate, batch size, number of layers, and units within the network to reach a combination that produces the best model with maximum accuracy and generalization abilities.

Furthermore, the study will look further into alternate techniques of segmenting continuous sensor data into windows by testing different window sizes and overlap strategies to see how these affect model performance. That is finding the optimal length of the window that balances capturing sufficient temporal information with computational efficiency.

The paper will test which characteristics extracted from sensor data are relevant to understand their impact on classification performance. In this study, different strategies for supervising the training process will be investigated in order to help the model focus on the relevant parts of the input sequences; the system will test their effectiveness in improving classification accuracy and generalization of the model. The study will cover an in-depth evaluation of the developed models with performance metrics and compare the tuned models with the baseline models to clearly indicate gains in improvements from tuning of hyper-parameters and other employed optimization techniques.

2. Literature Review

2.1. Existing Work on Activity Recognition

The field of Human Activity Recognition has gained significant momentum with the advent of machine learning and deep learning techniques. HAR involves identifying specific physical activities such as walking, running, and sitting, based on data collected from sensors typically embedded in wearable devices [8]. These devices generate vast amounts of time-series data that reflect the user's movement and physiological states [9].

Early works on Human Activity Recognition were based on conventional machine learning tools, with the most commonly used being Support Vector Machines, k-NN, and Decision Trees [10]. All of these models heavily depend on feature engineering to extract relevant descriptive features from sensor data. However, sensor data is characterized by high dimensionality and noise, which poses significant challenges to the solutions provided by the aforementioned methods; their performance often degrades in terms of generalization ability when encountering unknown data, especially when dealing with complex activities.

Instead of feature extraction, deep learning was invoked to replace this process by automatically extracting features from raw sensor data. For these reasons, CNNs have found wide applicability in both the efficiency in extracting and in giving rise to relevant spatial hierarchies [5]. CNNs are excellent at finding local dependencies and patterns for data gathered from sensors, making them very appropriate for accelerometer and gyroscope signal processing. CNNs outperform the classic models when performing multi-modal activity recognition, with the use of classical models when several sensor modalities [5].

RNNs, particularly LSTM networks, have also given very promising results in HAR [11]. LSTM networks are good at learning dependency through time within a sequence of data. The recognition of activities is most important over some time span. Combining CNN with LSTM networks using the properties above produces a hybrid model that captures both the spatial and temporal features. It could help to improve the recognition performance on mobile-devices data. Different hybrid variants have been proposed in network settings. For example, a Convolutional and Recurrent Neural Network (CRNN) was used to jointly model the spatial and temporal dependencies in sensor data, to boost the robustness of HAR systems [12]. In this regard, the

hybrid models have been found particularly effective for real-time activity recognition challenges for which necessity for speed, along with the accuracy that is of the utmost importance.

However, a number of challenges still remain for HAR, especially in terms of model generalization across users and environments. A strand of recent work deals with investigations on personalization techniques by fine-tuning the models to users, indicative of personal variations in the patterns of activities [10].

2.2. Hyperparameter Tuning in Machine Learning

Hyperparameter tuning is a leading factor in the route to obtaining a better machine-learning model. Unlike model parameters, which are learned during training, hyperparameters remain constant and define the algorithms of the model. They can include configurations such as the learning rate, batch size, number of layers, or number of units in a layer of a neural network. Proper tuning of hyperparameters enhances model performance in terms of accuracy and generalization. Poor choices lead to suboptimal performance, overfitting or underfitting.

Traditional sampling methods include grid search and random search [7]. Grid search recycles predefined grids of hyperparameters with exhaustive performance evaluation. However, this approach can be very computing-intensive, especially when modeling with deep learning optimization that uses large hyperparameter spaces. In contrast, random search simplifies hyperparameter combinations by random sampling [7]. It appears more effective in simpler cases than the exhaustive search, providing better results.

Hyperparameter optimization has relied heavily on techniques such as Bayesian Optimization [13]. Techniques like the Tree-structured Parzen Estimator model the conditional distribution between hyperparameters and the performance metric while guiding the search process towards more promising regions of the hyperparameter space [14]. This allows for more effective exploration since evaluating each configuration can be costly.

Tools like Optuna have made implementing advanced techniques for hyperparameter tuning easier. This is made possible by its flexibility and efficiency, demonstrated by its ability to dynamically adjust the search space according to previous trials. Optuna can use algororithms like TPE, Random Sampler, and Quasi-Monte Carlo (QMC) [14]. Other optimization techniques can be integrated. Studies [15] showed that applying a Bayesian Optimization with Gaussian Process (GP) in tuning hyperparameters could improve model performance, which can be achived in Optuna hyperparameter search.

Hyperparameter tuning is particularly crucial in the field of HAR due to the variability in sensor data and the complexity of activities. Models require to balance flexibility and robustness. Even minor tuning of model's hyperparameters could improve the generalization capabilities [13].

3. Dataset Overview

3.1. Dataset Description

The MHEALTH dataset consists of motion and vital signs data from ten subjects with diverse profiles, performing twelve physical activities. Data was recorded by sensors placed on the chest, left ankle, and right wrist. Readings include twenty-four features, one of which is the label.

The classes are:

`0`: *Null*,

`1`: *Standing still*,

`2`: *Sitting and relaxing*,

`3`: *Lying down*,

`4`: *Walking*,

`5`: *Climbing stairs*,

`6`: *Waist bends forward*,

`7`: *Frontal elevation of arms*,

`8`: *Knees bending (crouching)*,

`9`: *Cycling*,

`10`: *Jogging*,

`11`: *Running*,

`12`: *Jumping front & back*.

The dateset was retrieved from UCI archives [16].

3.2. Data Collection and Preparation

The data collection process was conducted by the publishers [6]. The dataset consists of Accelerations, ECG, Magnetometer, and Gyroscope readings. The senors recordings are visualized and their relevance is discussed in the this chapter in Section 3.4. Each subject was equipped with three devices that included sensors:

- *accelerometers* that measure the acceleration of motion, explored in ;
- *ECG sensors* recording the electrical activity of the heart,
- *magnetometers* measuring magnetic forces,
- *gyroscopes* that measure orientation and the angular velocity.

The publishers conducted additional processing steps, synchronization, noise filtering. The data was cuptered by mHealth Framework [17] providing functionalities for abstracting resources and communication, acquiring biomedical data, extracting health-related knowledge, storing data persistently.

The dataset was partially preprocessed by the publishers but required further preparation for analysis, including label refinement, structuring the data into distinct activity sessions, and preserving the chronological order of sensor readings.

Labels such as Standing still or Lying down already represented inactivity, therefore, records assigned to `0`, which did not correspond to any particular physical activity, were trimmed to focus the research on relevant activities. This decision helped maintain the integrity of the dataset by including only data associated with well-defined classes.

The continuous sensor data was broken down into sessions representing activities. This division was done by recognizing changes in activity labels; whenever a new label appeared, a new session automatically started, ensuring that the data within each session belonged only to one activity.

This was accomplished by automatically numbering each recording step of all readings with a progressively increasing sequence number, ensuring that chronological order was maintained throughout the sensor processing.

3.3. Database Setup

The database setup for this study is proposed to enable efficient access to the collected sensor data. It allows accessing large time-series data in a structured format for diverse experiments and hyperparmeter tunnings with diffrent window shifts and diffrent model architectures. This setup was proposed to address memory management concerns during training when operating on large number of data slices. Used structure ensures the system remains efficient, as it scales with larger amounts of data. This setup uses technologies:

- ***PostgreSQL-*** Database Management System in which Sensor Data are stored [18],
- ***SQLAlchemy-*** tool for Object-Relational Mapping for database interaction [19],
- ***TensorFlow-*** integrated machine learning framework for processing [20].

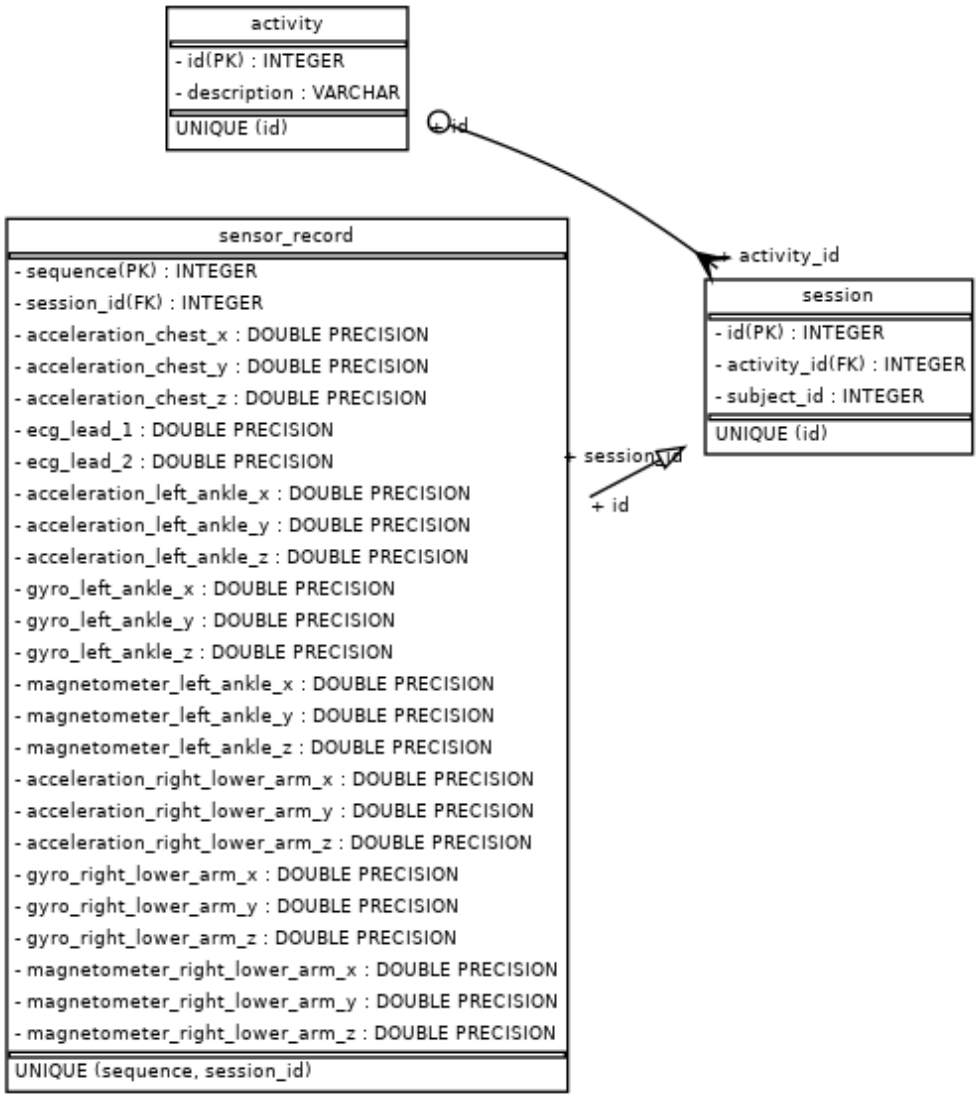

Figure 3.1: Database Schema for MHEALTH Study

As shown in Figure 3.1, the database schema defines three tables: the `Activity` table, which contains all observed activities, identified and labeled; the `Session` table, which provides information about individual activity sessions tied to an activity and subject; the `Sensor_Record` table, which holds sensor data, ordered entries associated with each session.

The database setup ensures a smooth flow of data throughout the training phase. System generates data segments that are fed to the model on the fly. This approach reduces memory usage by generating only the necessary data blocks at a given time, preventing over-allocation of RAM. TensorFlow integration ensures streaming data processing, providing the data is placed in correct slices, batches, prefetched and shuffled.

The time-series data can be accessed by braking down the dataset into windows of the same size. New overlapping segments could be generated by slightly offsetting where each window begins. This could multiply the avaible training data, producing more varied dataset and can help the model to learn more effectively. This setup enables extensive experimentation by adjusting window sizes and shift distances to extract multiple data segments, which is especially useful for increasing model accuracy when working with limited data.

3.4. Sensor Data and Features Study

All records assigned to subjects have been aggregated to show how the data is diversified and distributed. Table 3.1 shows the total number of records of sensor data belonging to each of the subjects who participated in the experiment.

Subject_ID	SensorData_Count
1	35174
2	35532
3	35380
4	35328
5	33947
6	32205
7	34253
8	33332
9	34354
10	33690
Total	**343195**

Table 3.1: Sensor Data Count per Subject

The sensor data was relatively evenly distributed across the subjects, in a total of 343,195 records. These make sure that there is balance in distribution so that, later, when the analysis and models ensue successively, one subject's bias is not projected.

It is essential to note the duration and time of different activities recorded in the dataset. Figure 3.2 shows overall duration of various activities recorded in the dataset. All recorded activities were counted and transformed into a representation in minutes. The graph shows the distribution of counted activities.

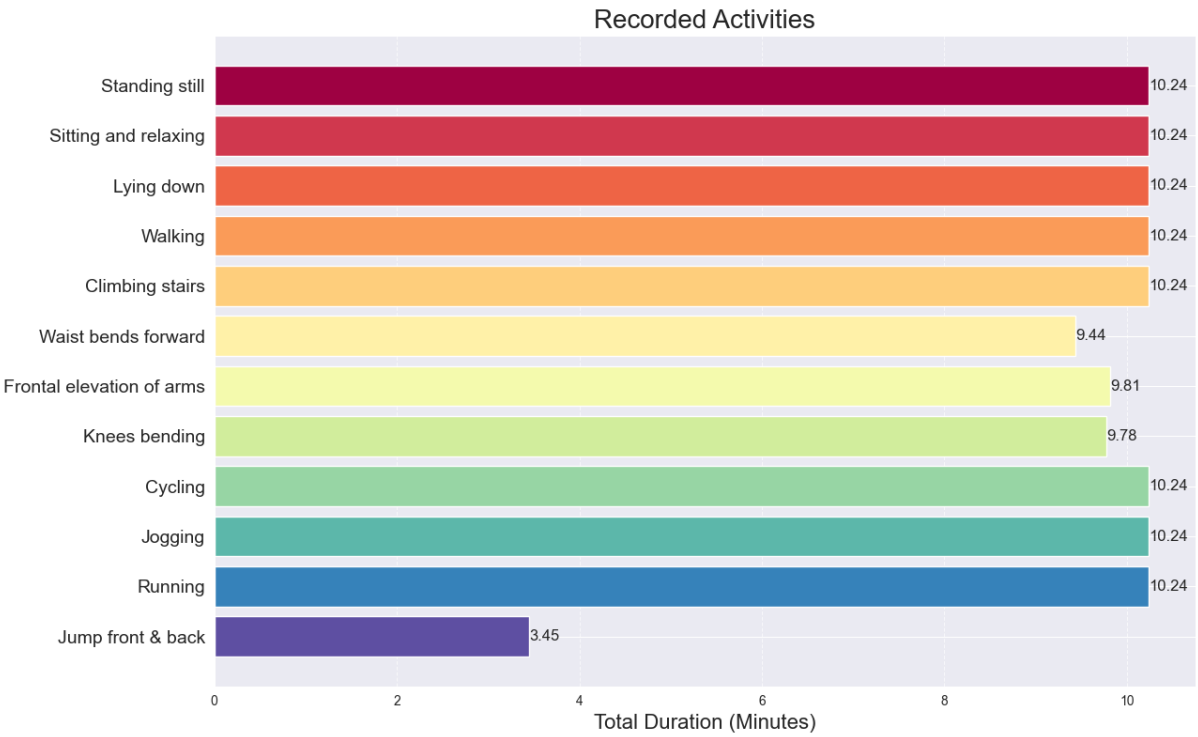

Figure 3.2: Recorded activities graph

This graph shows that most activities have a duration of 10.24 minutes, with the exception of *Waist bends forward*, *Frontal elevation of arms*, *Knees bending*, which have slightly lower durations. However, records labeled as *Jump front & back* provide only 3.45 minutes of recording. The overall uniformity in activity duration ensures comprehensive coverage of various physical movements in the dataset.

3.4.1. Acceleration Readings

Accelerometers measure the acceleration of motion. An accelerometer measures acceleration, tilt, and vibration. The graph below presents the average magnitude of acceleration for the recorded activities using the recordings of chest sensors, showing the relative intensity of activities by calculating the mean acceleration over the x, y, and z axes and ordering the activities by their intensity levels. Figure 3.3 plots the different activities against their average magnitude of acceleration.

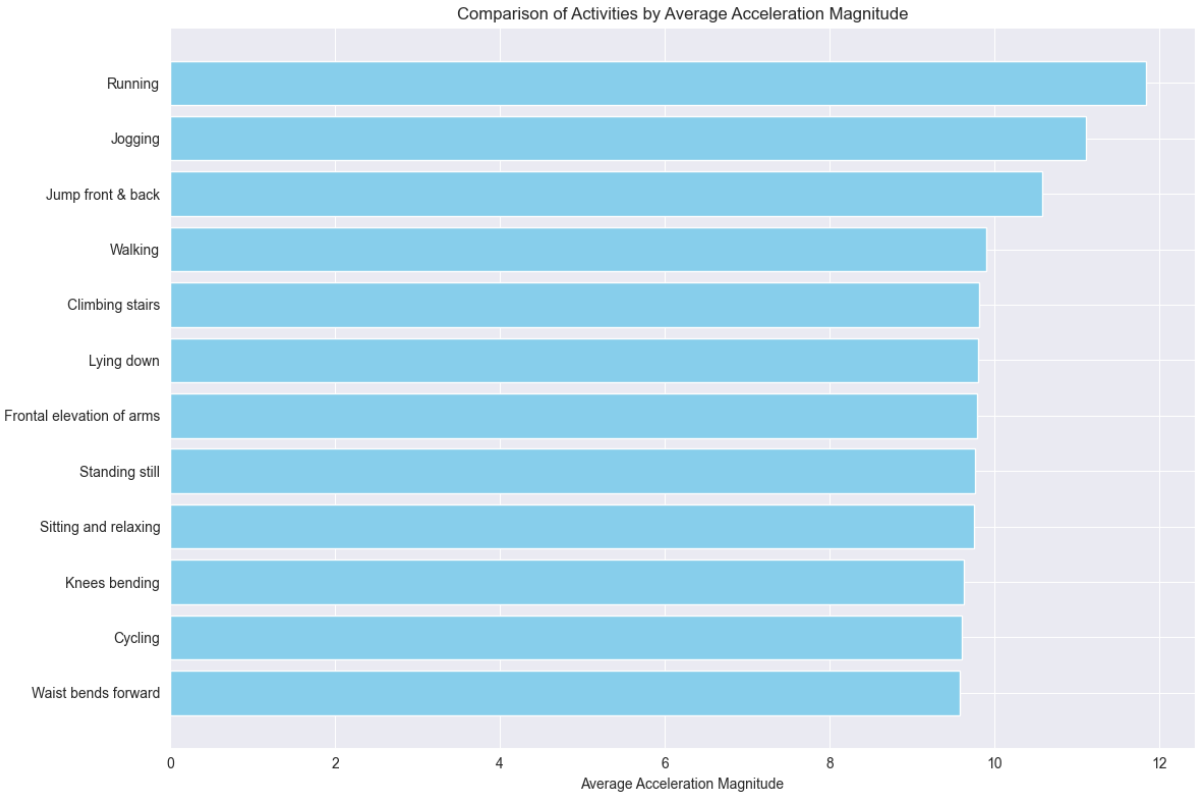

Figure 3.3: Comparison of Activities by Average Acceleration Magnitude

This straightforwardly shows that running, jogging, and jumping are activities with higher values of average acceleration, thus greater physical effort, while standing still and sitting have low magnitudes of acceleration. These simple observations help in classifying activities in terms of high-intensity versus low-intensity activities.

For each activity the mean acceleration for the left ankle and the right lower arm were calulated. The graph below shows a comparison of the mean acceleration with respect to different activities. The graph shows these means for every activity. Activities can be easily compared with one another, and their physical intensity captured by sensors located at these two anatomical regions. Average magnitude accelerations from these sensors on the left ankle and right lower arm were computed to perform an analysis of how different parts of the body move during different activities. This comparison is shown in Figure 3.4.

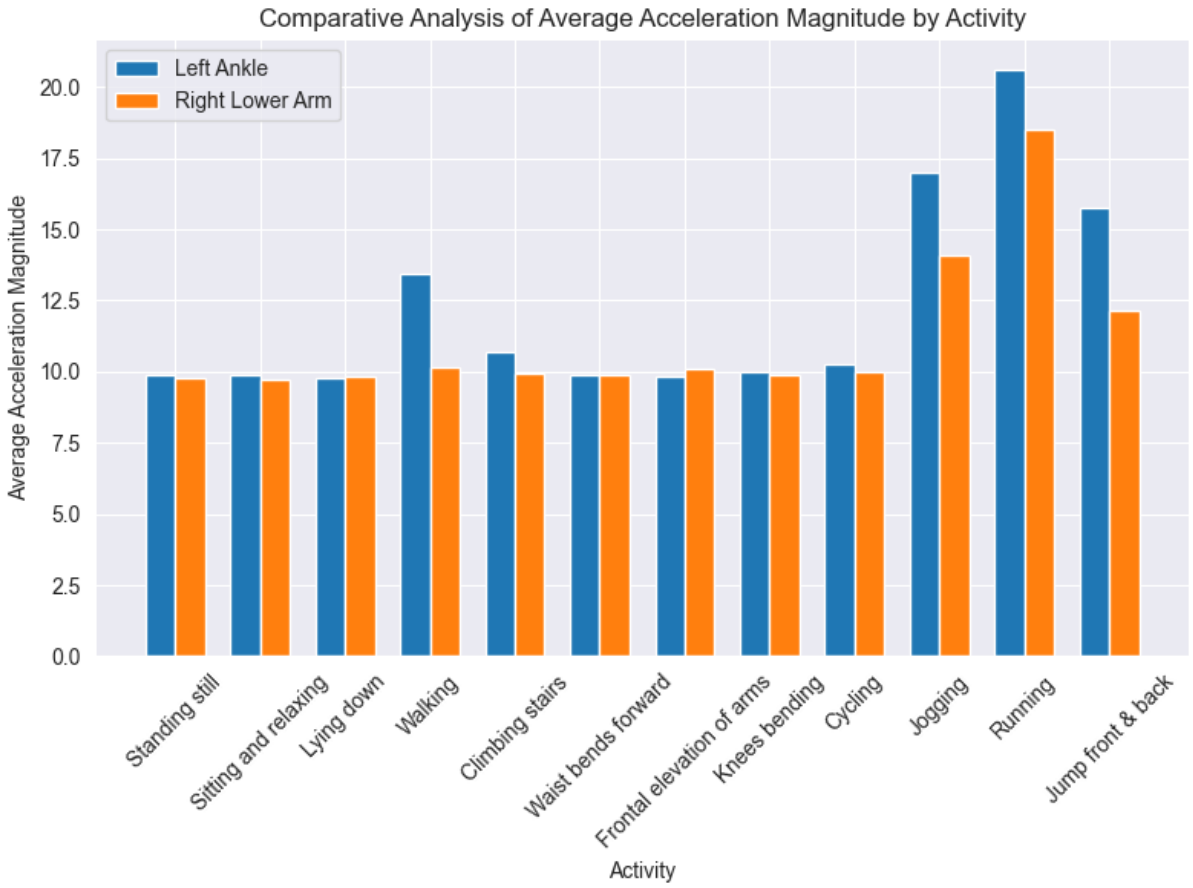

Figure 3.4: Comparative Analysis of Average Acceleration Magnitude by Activity

The bars represent the average magnitude of acceleration, where one corresponds to the left ankle and another to the right lower arm. In this way, it will be easier to notice differences in motion patterns between both body parts during different activities. The comparison is useful in arriving at some preliminary observations about how different activities engage different body parts at varying levels of intensity. These results show quite large differences in locomotion patterns according to the two sensor positions. For example, walking and running activities exhibit stronger acceleration at the left ankle compared to the right lower arm, indicative of typical biomechanical characteristics for these activities.

Feature engineering and multivariate analysis are critical to understanding the relationships between measurements from different sensors. Figure 3.1 illustrates the correlation heatmap of acceleration measurements.

This analysis shows significant differences in movement patterns between the two sensor positions. For example, activities like walking and running show higher acceleration at the left ankle compared to the right lower arm, reflecting the natural biomechanics of these activities.

Investigating the relationships between different sensor measurements is crucial for feature engineering and multivariate analysis. Figure 3.5 shows a correlation heatmap of the acceleration measurements.

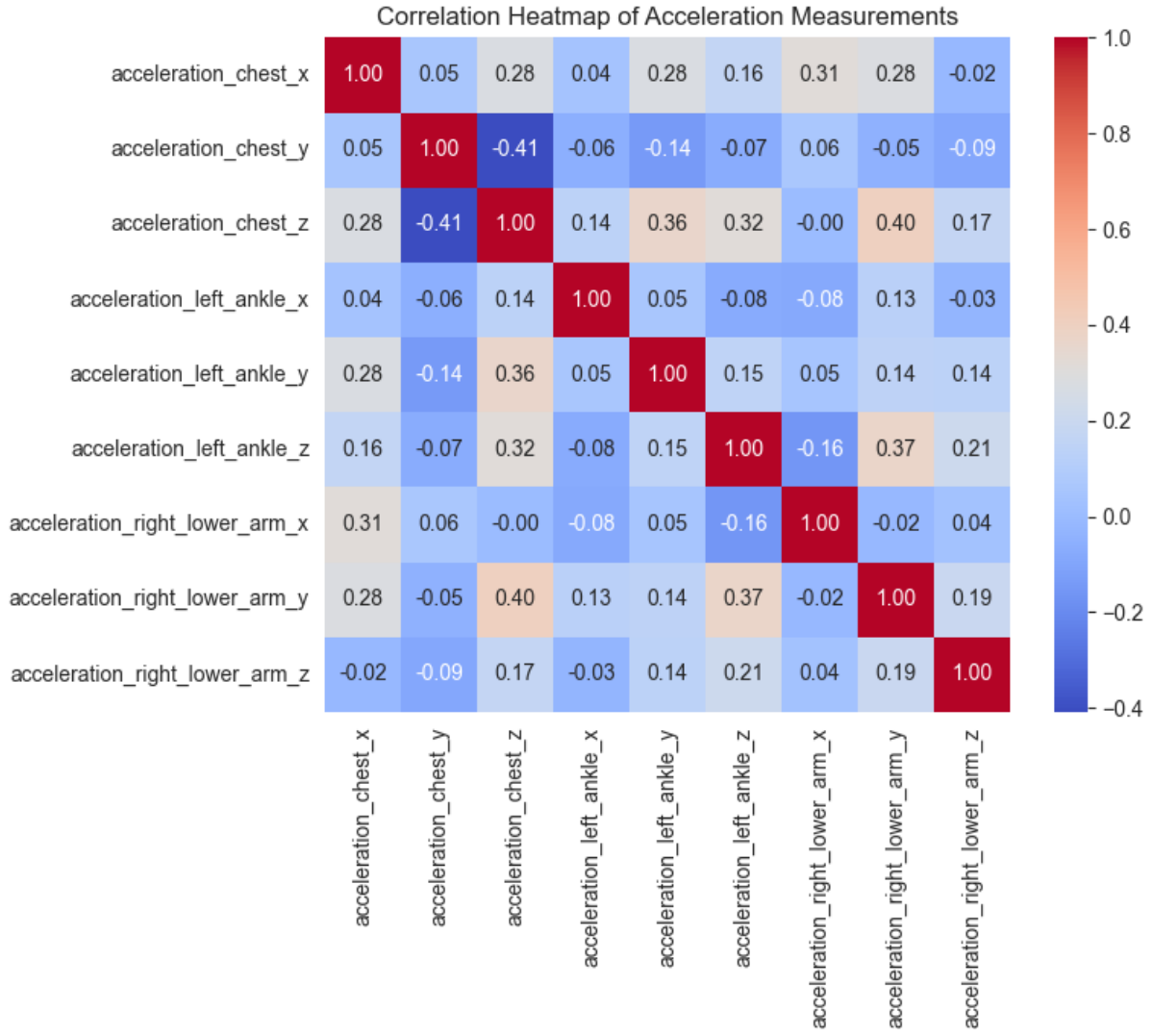

Figure 3.5: Correlation Heatmap of Acceleration Measurements

The heatmap indicates the degree of linear relationship between pairs of acceleration measurements from different sensor axes and positions. The heatmap summarizes the correlation for a number of different acceleration measurements from various axes and positions. Very little

correlation was found between the chest, left ankle, and right lower arm sensors, meaning that most of the movement detected by these sensors is independent, with each sensor picking up distinct patterns of motion. Moderate correlation between different axes at the same sensor position indicates some interdependence in the movement that these axes detect. The negative correlations, such as those between the chest y-axis and chest z-axis, point to movement patterns where an increase in one axis's acceleration corresponds to a decrease in another's.

3.4.2. ECG Readings

ECG readings where ploted for each activity over time to estimate their physiological effects. The two charts 3.6 and 3.7 represent average ECG traces from two different leads over a time signature of each activity.

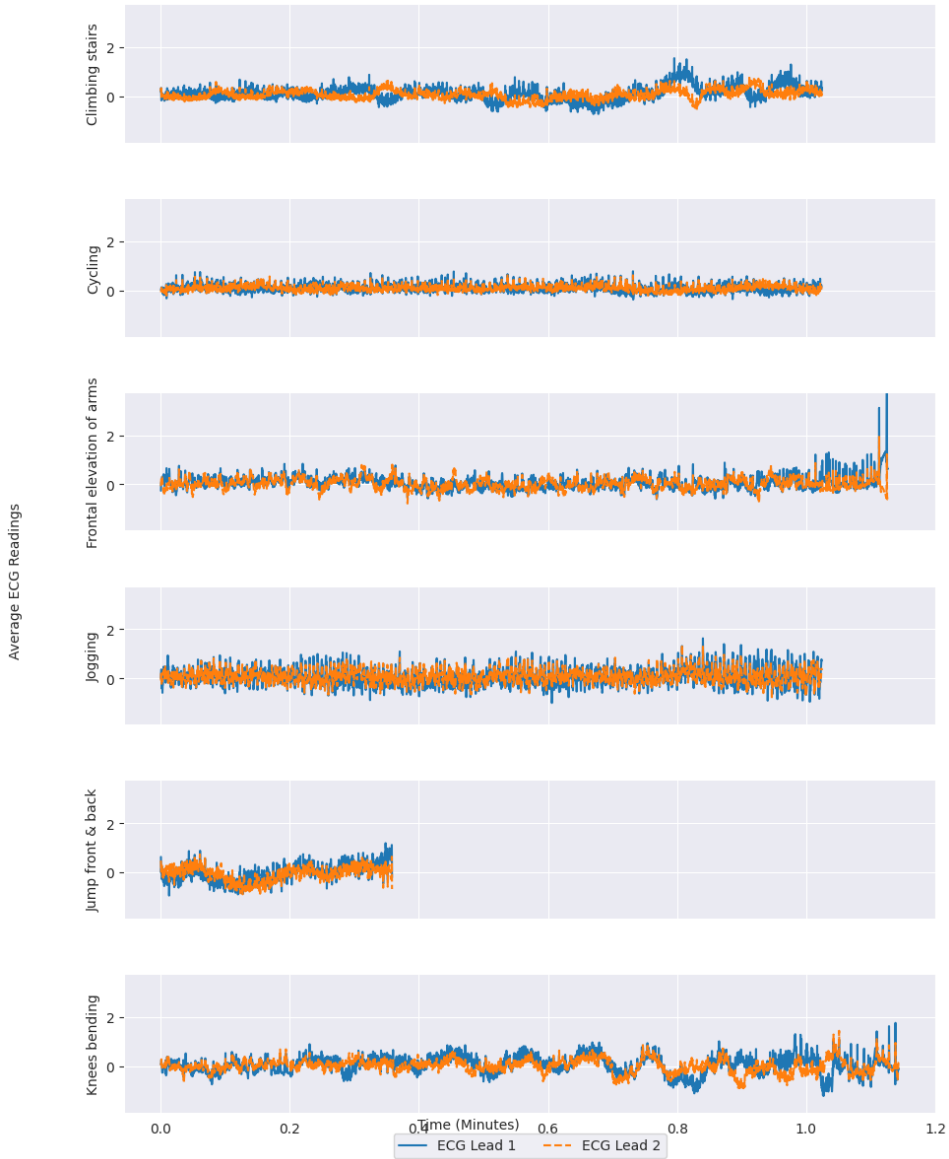

Figure 3.6: ECG Readings Analysis Across Activities - part 1

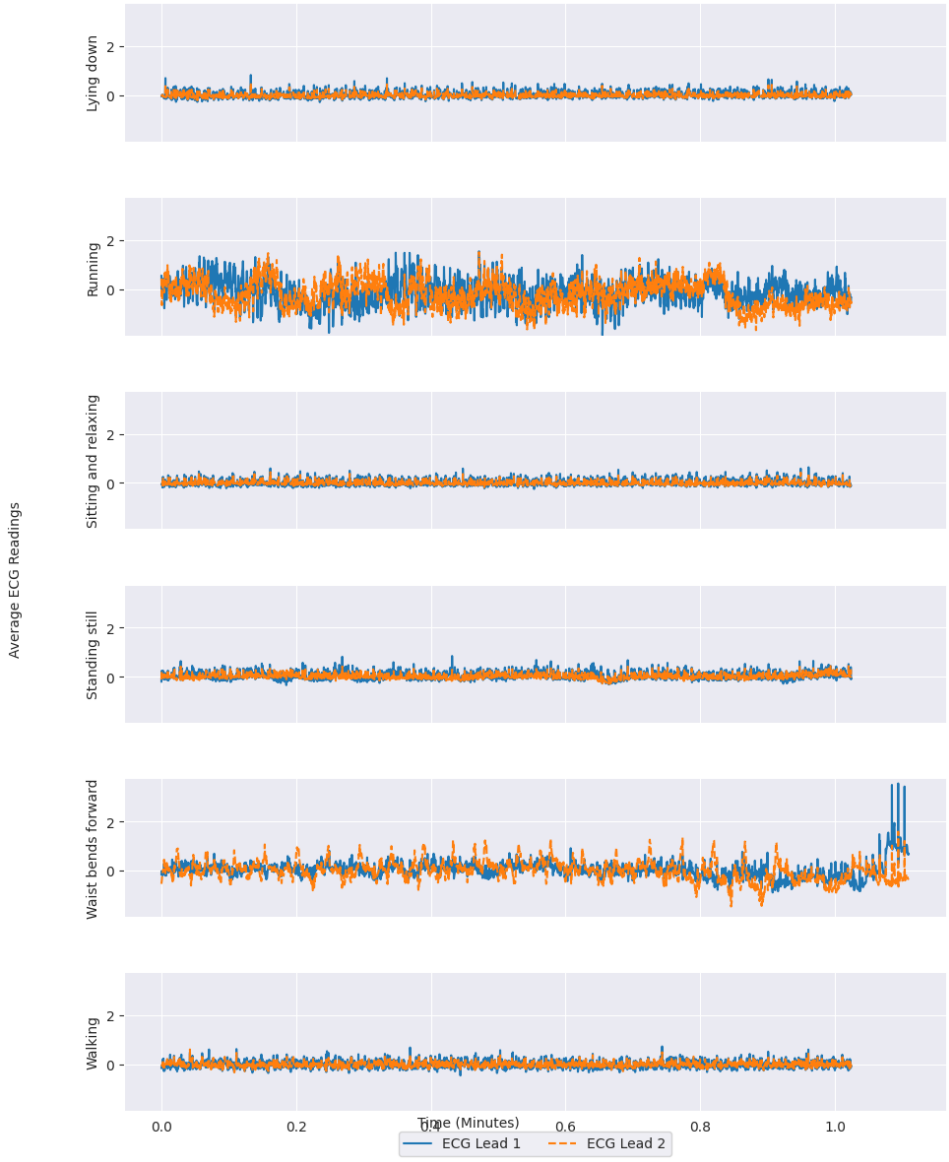

Figure 3.7: ECG Readings Analysis Across Activities - part 2

Each activity is represented with ECG readings, showing the minute-by-minute changes reflected in both Lead 1 and Lead 2 of the ECG. This chart provides the possibility of correlating the cardiovascular responses in these activities and highlights cardiac activity profiled with regard to the intensity level of each particular activity.

Each subplot in this figures represents a specific activity, showing how the ECG readings fluctuate during its performance. For instance, activities like running and jogging show more variability in ECG readings, indicating higher cardiovascular activity, while more sedentary activities like sitting and lying down show less variability.

3.4.3. Magnetometer Readings

The magnetometer readings provide insight into the magnetic field variations during different activities. The average magnetometer readings on the X, Y, and Z axes for all recorded activities are shown in Figure 3.8 (for sensors placed at the left ankle and right lower arm):

Reading magnetometer readings provide information on magnetic field changes from different activities. This could be used to deduce orientation and movement. Figure 3.8 shows the mean magnetometer readings in the X, Y, and Z axes across different activities (see sub-section Sensors).

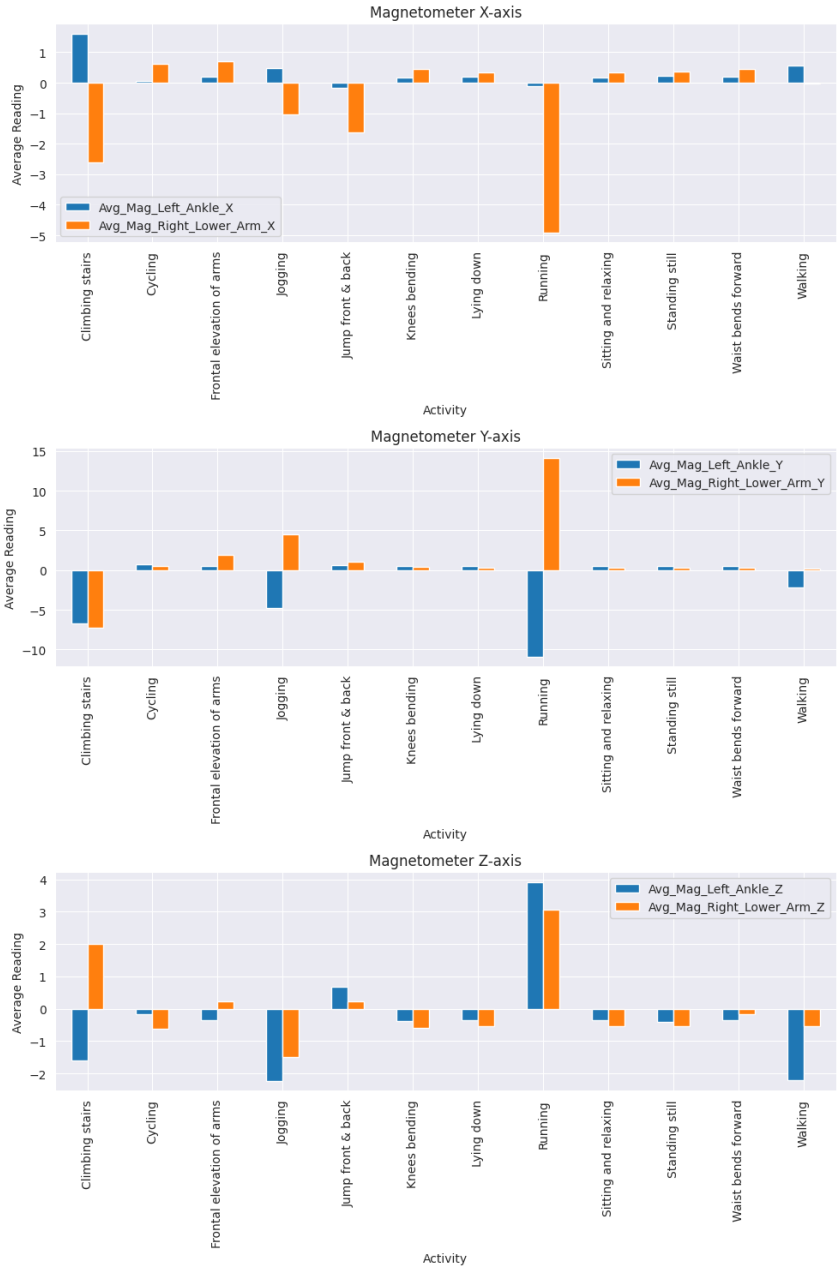

Figure 3.8: Average Magnetometer Readings - X, Y, Z axis

The visualizations show that different physical activities yield unique values from the magnetometer. For example, running and jogging show more variability in magnetometer readings, which points to dynamic movement, orientation changes, whereas static activities like sitting and lying down do not have the same amount of signal.

In addition, magnetometer and accelerometer data allow us to evaluate the correlation between different types of movements. Figure 3.9 depicts the correlation heatmap between magnetometer and accelerometer readings.

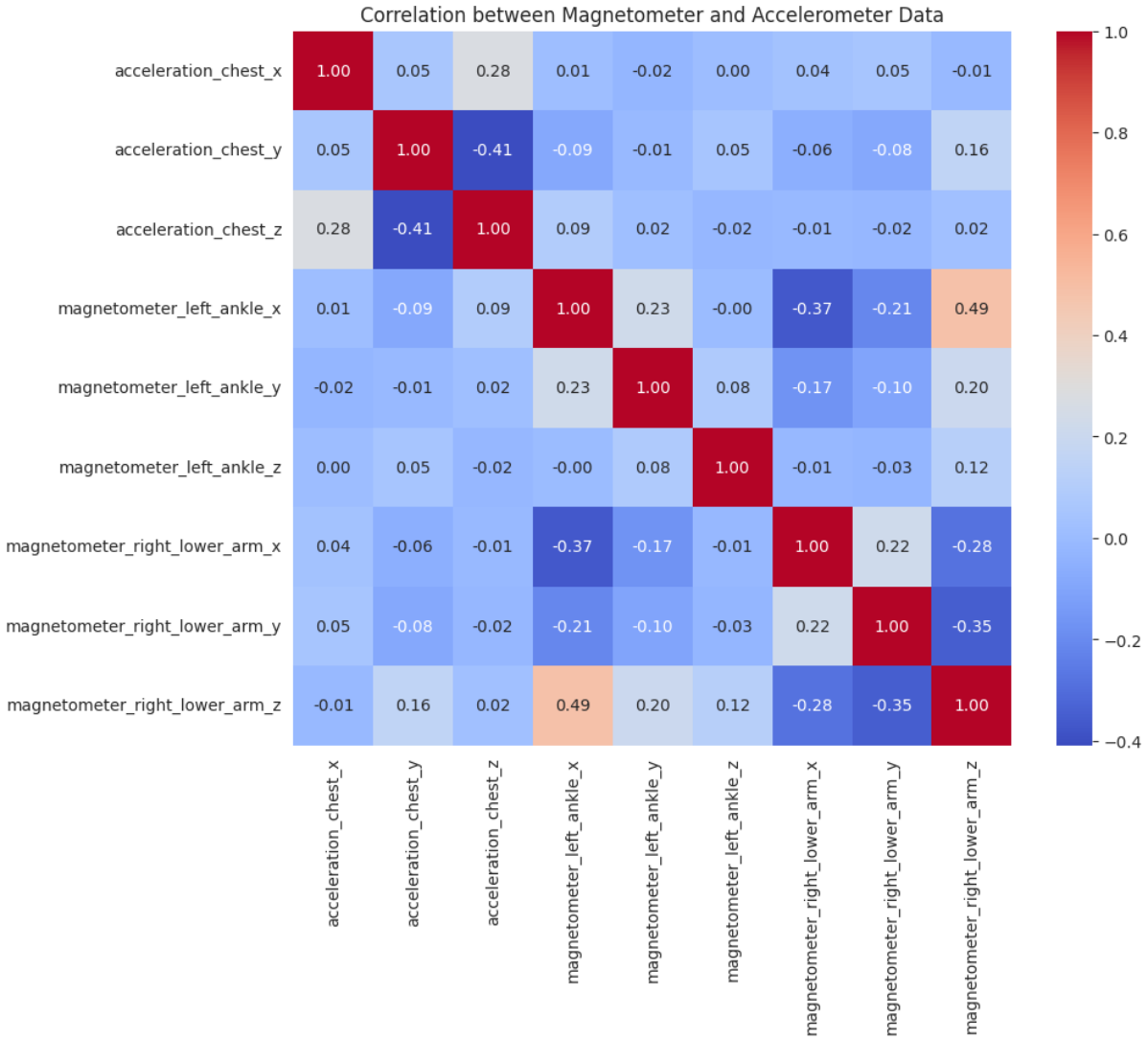

Figure 3.9: Correlation Heatmap between Magnetometer and Accelerometer Data

It shows that certain axes of the magnetometer and accelerometer sensors are closely related, the most closely are: Magnetometer Right Lower Arm Z with Magnetometer Left Ankle Z, Magnetometer Right Lower Arm X with Magnetometer Left Ankle Y, and Magnetometer Right Lower Arm Z with Magnetometer Right Lower Arm Y. This identifies potential relationships that can be utilized later in modeling process.

3.4.4. Gyroscope Readings

Accelerometer determines the acceleration of different body parts during various activities, whereas gyroscope data provides information about angular velocity. The correlation heatmap of gyroscope readings and the average gyroscope readings by activity are shown in Figures 3.10 and 3.11.

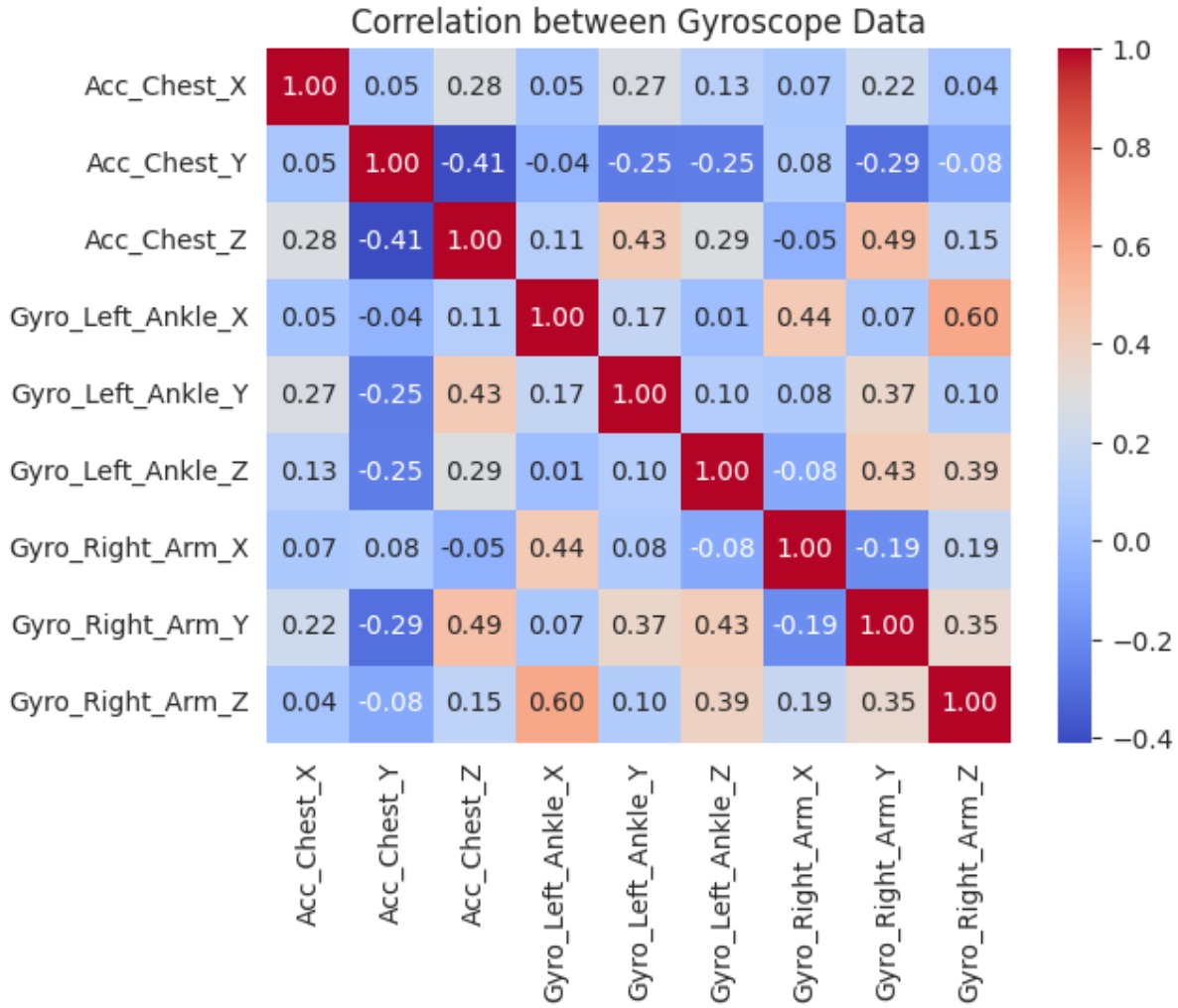

Figure 3.10: Gyroscope Correlation Data

Figure 3.10 shows Correlation Heatmap of the gyroscope data, presents inter-correlations from different body parts, how closely correlated each pair of gyroscope measurements in different sensor axes and body parts are. It shows, for example, a high positive correlation of 0.60 between the gyroscope readings from the left ankle (X dimension) and the right arm (Z dimension); this may represent how the movement on the X-axis of the left ankle is mirrored by the Z-axis of the right arm. This could be due to symmetric patterns of movement or coordinated activities involving both limbs.

Several pairs of gyroscope data also have very low correlations, indicating these movements are largely independent: the correlation between accelerometer reading from the chest dimension and gyrometer from the right arm (both dimension X) is close to zero, showing minimal to no direct relationship between these two axes.

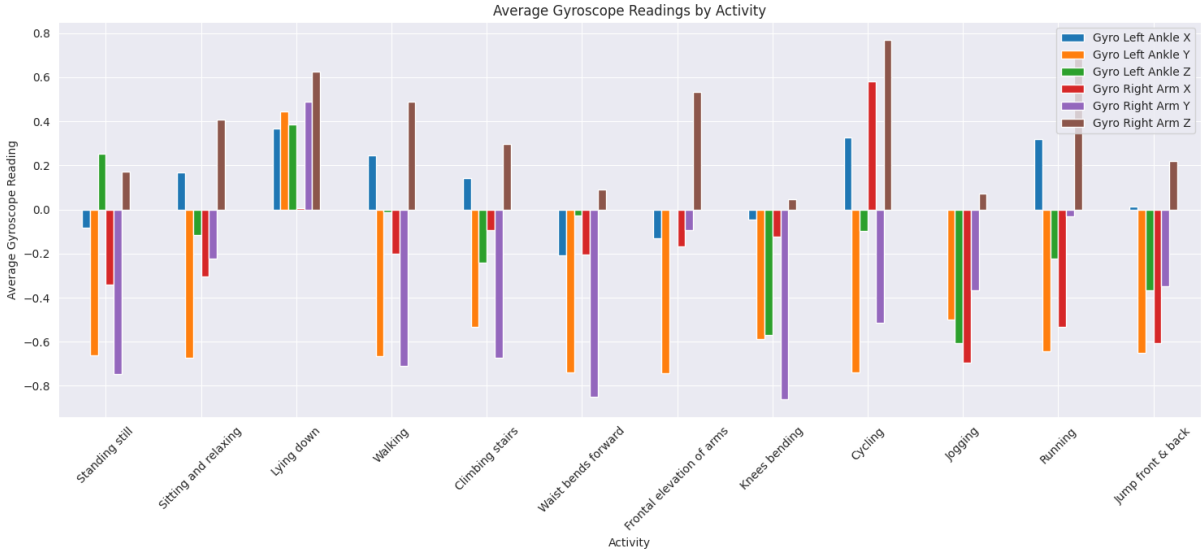

Figure 3.11: Average Gyroscope Readings per Activity

The comparison of average gyroscope readings by activity demonstrates significant differences in angular velocity across activities. High angular velocities are observed for dynamic activities such as running and jumping, relative to static ones like sitting or standing still. This distinction makes it easier to classify and understand various activities through gyroscope data.

4. Methodology

4.1. Model Selection

Different neural network architectures are discussed in this paper to solve time series classification task; these range from the purely recurrent networks through hybrids using both convolutional and recurrent layers. Dense layers at the end of the network are also proposed. All the discussed architectures performances on the studied dataset are reported in the Section 5.3.

4.1.1. Recurrent Neural Networks (RNN)

Unlike the standard neural networks, RNNs are designed to contain a hidden state that helps them in effectively modeling and capturing dependencies present in sequential data, which makes them very good in recognizing patterns and trends over time. Traditional RNNs often exhibit the vanishing and exploding gradient problems, which set a ceiling on their ability to learn long-term dependencies. Classifications of these types of problems are being tackled with more complicated architectures, such as Long Short-Term Memory and Gated Recurrent Units. The enhanced models include mechanisms that control the flow of information across time steps, thus giving better results in tasks related to time series classification.

The most basic RNN cell updates its hidden state $\mathbf{h}_t$ based on the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h) \tag{4.1}$$

where:

- $\mathbf{h}_t$ is the hidden state at time step t,
- $\mathbf{x}_t$ is the input at time step t,
- $\mathbf{h}_{t-1}$ is the hidden state from the previous time step $t-1$,

- $\mathbf{W}_h$ is the weight matrix for the input $\mathbf{x}_t$,
- $\mathbf{U}_h$ is the weight matrix for the previous hidden state $\mathbf{h}_{t-1}$,
- $\mathbf{b}_h$ is the bias vector,
- $\tanh$ is the activation function used in the RNN cell.

Long Short-Term Memory (LSTM)

The LSTM networks were designed in a way to defeat the vanishing gradient problem faced in training traditional RNNs [21]. An LSTM cell is made up of components, each of which plays a critical role in its ability to learn and remember things over time. These include an input gate, a forget gate, and an output gate.

Input Gate, depicted in (4.2), controls the quantity of new information from the current input $\mathbf{x}_t$ to inject into the state of the cell $\mathbf{c}_t$.

$$\mathbf{i}_t = \sigma\big(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i\big) \tag{4.2}$$

Forget Gate, as shown in (4.3), determines how much of the previous cell state, $\mathbf{c}_{t-1}$, it wants to keep.

$$\mathbf{f}_t = \sigma\big(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f\big) \tag{4.3}$$

The candidate cell, shown in (4.4), is the new content that would be added to the cell state if given the chance.

$$\tilde{\mathbf{c}}_t = \tanh\big(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c\big) \tag{4.4}$$

Cell State Update, depicted in (4.5), adds the new candidate state, gated by input, to the memory cell along with the previous state that is being updated by forget.

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \tag{4.5}$$

Output Gate, as shown in (4.6), controls the extent to which the cell state $\mathbf{c}_t$ should affect the hidden state $\mathbf{h}_t$.

$$\mathbf{o}_t = \sigma\big(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o\big) \tag{4.6}$$

The hidden state, represented in (4.7), can be considered the output of the LSTM cell and is used for further prediction or advanced in time.

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{4.7}$$

Gated Recurrent Unit (GRU)

The GRU is a simplified form of the LSTM, with the forget gate combined into the input gate as a single update gate, which makes it less complex, but it still retains the ability to model the dependencies of the sequence.

Update Gate, shown in (4.8), defines the amount of the previous state, $\mathbf{h}_{t-1}$, to be saved within the current state.

$$\mathbf{z}_t = \sigma\big(\mathbf{W}_z\mathbf{x}_t + \mathbf{U}_z\mathbf{h}_{t-1} + \mathbf{b}_z\big) \tag{4.8}$$

Reset Gate, as indicated in (4.9), controls how much of the previous hidden state $\mathbf{h}_{t-1}$ should be ignored in the computation of the new candidate hidden state.

$$\mathbf{r}_t = \sigma\big(\mathbf{W}_r\mathbf{x}_t + \mathbf{U}_r\mathbf{h}_{t-1} + \mathbf{b}_r\big) \tag{4.9}$$

Candidate Hidden State, computed using the reset gate, is described in (4.10) as another candidate for the new hidden state.

$$\tilde{\mathbf{h}}_t = \tanh\big(\mathbf{W}_h\mathbf{x}_t + \mathbf{U}_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h\big) \tag{4.10}$$

Hidden State Update, shown in (4.11), is modulated by an update gate, leading to a weighted sum of the last hidden state and candidate state.

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t \tag{4.11}$$

GRUs have fewer gates, which means faster training compared to other deep learning models, as demonstrated in studies [22], with lower computational costs while retaining the ability to model complex temporal patterns. Additionally, GRUs have achieved performance comparable to that of LSTMs in many applications, making them a practical and moderate baseline choice. Lastly, they are less complex, which reduces the risk of overfitting.

4.1.2. Convolutional Layers

Convolutional layers are the elementary units of convolutional neural networks. A feature map provides the specific locations of the features detected in the input and their associated strengths. The main advancement of convolutional neural networks is that they can learn to create numerous filters in a more automatic way, the number dependent on the training data and task, such as forming the input layer for classification [23]. It leads to very sparse features that respond anywhere in the input data. The convolutions' task is to apply filters to input time series data by identifying patterns, such as trends or cycles.

The output of a 1D convolutional layer is given by:

$$y_t = \sum_{i=1}^{k} x_{t+i-1} \cdot w_i + b \tag{4.12}$$

where:

- x_{t+i-1} is the input at time step $t + i - 1$,
- w_i are the weights of the convolutional filter,
- k is the size (length) of the filter,
- b is the bias term.

These features are instrumental in efforts geared towards improving classification accuracy when fed into recurrent layers [24].

Pooling Layers

Pooling layers reduce the size of feature maps derived from convolutional layers. They decrease computational complexity and can potentially reduce overfitting by down-sampling the data.

For max pooling, the output is:

$$y_t = \max(x_{t1}, x_{t2}, \ldots, x_{tn}) \tag{4.13}$$

where:

- y_t is the maximum value within the pooling window,
- $x_{t1}, x_{t2}, \ldots, x_{tn}$ are the input values in the pooling window.

For average pooling, the output is:

$$y_t = \frac{1}{n} \sum_{i=1}^{n} x_{ti}, \tag{4.14}$$

where:

- y_t is the average value of the inputs in the pooling window,
- n is the number of inputs within the pooling window,
- x_{ti} are the input values in the pooling window.

Pooling is useful because it allows for the creation of smaller feature representations that can be passed into recurrent layers, helping to improve overall model performance [25].

Adding Convolutional and Recurrent Layers

Combine convolution and recurrent layers to capture local feature extraction by CNNs and the temporal modeling capability of RNNs. This combination works very well for classification tasks on time series because both local patterns and long dependencies are an important value [26].

4.1.3. Dense Layers

Right after the stage of feature extraction and temporal modeling, the network ends in dense layers. Dense layers aggregate the features extracted by convolutional and recurrent layers and perform the final classification [27]. Dense layers, considering their usage in this study, leverage

the features learned by the model to make the final prediction.

The output of a dense layer is computed as:

$$z = \sigma(\mathbf{W}\mathbf{h} + \mathbf{b}) \tag{4.15}$$

where:

- z is the output of the dense layer,
- $\mathbf{W}$ is the weight matrix,
- $\mathbf{h}$ is the input feature vector from the previous recurrent or convolutional layer,
- $\mathbf{b}$ is the bias vector,
- σ represents the activation function, typically softmax for classification tasks.

Combining Convolutional, Recurrent, and Dense Layers

The architecture allows combining powerful convolutional, recurrent, and dense layers within time series classification. The exact convolutional layers extract local features; the recurrent layer models the dependencies in time while further dense layers classify extracted features. This layered architecture captures the local patterns and long-range dependencies for effective classification [28].

4.2. Performance Metrics and Loss Function

The loss function applied in this work is Sparse Categorical Cross-entropy, with metrics of Sparse Categorical Accuracy and Sparse Top-K Categorical Accuracy. These two metrics, together with the loss function, were chosen and very well adapted to multi-class classification tasks represented in the dataset.

4.2.1. Sparse Categorical Crossentropy Loss

One of them is the Sparse Categorical Crossentropy loss function, which is designed to be used in multi-class problems, in particular with deep learning models that process a huge amount of data [11, 5]. The said loss does not require one-hot encoding, it is suitable for integer labels in the studied dataset.

The loss for a particular training example is calculated by:

$$\mathcal{L}(\theta) = -\log\left(p_\theta(y|x)\right) \tag{4.16}$$

where:

- $p_\theta(y|x)$ is the estimated probability of the true class y given the input x,
- θ refers to the model's parameters.

For a dataset with N samples, the total loss is the average of individual losses:

$$\mathcal{L}_{\text{total}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\log\left(p_\theta(y^{(i)}|x^{(i)})\right) \tag{4.17}$$

This loss function enforces the model to have a high probability on the correct class for each input. In its implementation, it penalizes wrong predictions and guides the model toward increasing its classification accuracy [29].

4.2.2. Sparse Categorical Accuracy

Sparse Categorical Accuracy defines the rate of correct predictions made by the model [31]. It is mainly useful in cases of multi-class classification problems, where each instance belongs to only one class. This metric gives an easy measure of how often the model's top prediction matches the true label, which is critical for interpreting the model's overall performance within HAR. Common in the area of human activity recognition tasks [30].

Sparse Categorical Accuracy is defined as:

$$\text{Sparse Categorical Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}\left(\hat{y}^{(i)} = y^{(i)}\right) \tag{4.18}$$

where:

- N represents the total number of predictions,
- $\hat{y}^{(i)}$ is the predicted class for the i-th instance,
- $y^{(i)}$ is the actual class of the i-th example,
- $\mathbf{1}(\cdot)$ is 1 if the predicted class matches the actual class, 0 otherwise.

4.2.3. Sparse Top-K Categorical Accuracy

Sparse Top-K Categorical Accuracy checks whether the true label is among the top K predicted classes. The utility of this measure arises from the fact that models could often output several possible classes, and sometimes it is necessary to check whether the correct class is among the top predictions [1].

It follows the definition of Top-K Sparse Categorical Accuracy:

$$\text{Sparse Top-K Categorical Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1} \left(y^{(i)} \in \hat{y}_k^{(i)} \right) \tag{4.19}$$

where:

- N is the total number of predictions,
- $\hat{y}_k^{(i)}$ is the set of top K predicted classes for instance i,
- $y^{(i)}$ is the true class of the i-th instance.

This could be helpful in this research as the activities discovered may have closely congruent sensor patterns. The metric ranks the correct activity within the top K predictions [11].

4.2.4. Application of Metrics to the Study

These metrics can thus be applied to ascertain how models can generalize well over unseen data and how good they are in discriminating different activities. These metrics guide the optimization process of training in a manner where the models emphasize improvement not just in the accuracy of the top prediction but also in the relevance of the top K predictions.

Emphasizing these two metrics ensures that the models developed are not just accurate but also robust against scenarios where activities have very similar characteristics. The evaluation hence becomes critical later for the success of the study, in which recognition against a wide range of physical activities will be done accurately and reliably.

4.3. Optimizers

Optimizers are central algorithms to the process of training a neural network that search for reducing an error by adjusting weights and biases, among other parameters [29]. By adjusting these, optimizers can actually set how far and fast the system is in the process of learning, allowing it to converge toward an optimal solution. Choosing a suitable optimizer is of utmost significance, particularly in tasks like Human Activity Recognition (HAR) involving patterns across time. For the model to learn from such data effectively and efficiently, it needs to be

optimized well.

This work considers a diverse set of very popular optimizers, all widely designed to meet challenges in HAR:

- SGD that adds randomness (stochasticity) into the process, this can help the model escape more often from a local minimum and provide more insights into the fundamental behavior of the model;
- RMSProp algorithm that is very effective in scenarios relating to non-stationary data, such as HAR, where activity patterns vary greatly [32];
- Adam, Adaptive Moment Estimation, with robust optimization by estimating the first-order and second-order moments of the gradients, which can help with learning complex temporal dependencies in HAR [33];
- Adamax, an Adam variant with its update rule modified to handle massive or sparse gradients better and more robustly, providing additional stability during HAR model training [33];
- Nadam, Nesterov-accelerated Adaptive Moment Estimation that introduces Nesterov momentum to speed up convergence that could make it particularly effective in HAR tasks where scenarios change rapidly [34].

4.3.1. Gradient Descent

Gradient Descent is a baseline optimization algorithm that moves the model parameters in the negative gradient direction with respect to the loss function. Although sometimes it is applicable, gradient descent can be slow to converge, especially if there are noisy gradients or suboptimal learning rates [35].

The gradient update rule for gradient descent is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t) \tag{4.20}$$

where:

- θ_t are the parameters at iteration t,
- η is the learning rate,
- $\nabla_\theta L(\theta_t)$ is the gradient of the loss with respect to the parameters.

4.3.2. SGD (Stochastic Gradient Descent)

In gradient descent, the parameters are modified according to the gradient computed from the entire dataset. In the stochastic gradient descent version of GD, the parameters are updated using a single data point or mini-batch rather than the full dataset. This introduces some randomness into the optimization process that can be beneficial to ensure the model does not get stuck in local minima. The update rule for SGD is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; x_i) \tag{4.21}$$

where:

- x_i is a random data point or mini-batch chosen from the training set,
- θ_t are the parameters at iteration t,
- η is the learning rate,
- $\nabla_\theta L(\theta_t; x_i)$ is the gradient of the loss with respect to the parameters for x_i.

4.3.3. RMSProp (Root Mean Square Propagation)

RMSProp seeks to solve the varying gradient problem in optimization. RMSProp keeps a moving average of the squared gradients, which helps adapt the learning rate for each parameter at every instance, hence producing more steady and adaptive behavior compared to some other methods like SGD.

The update rules for RMSProp are:

$$\begin{aligned} E[g^2]_t &= \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{4.22}$$

where:

- $g_t = \nabla_\theta L(\theta_t)$ is the gradient of the loss function at each time t,
- ρ is the decay factor (usually set to 0.9),
- ϵ is a small constant to avoid division by zero,
- η is the learning rate.

RMSProp helps to stabilize the learning rate and is particularly useful for non-stationary objectives [32].

4.3.4. Adam (Adaptive Moment Estimation)

Adam is another variant of RMSProp that computes the first moment (mean) with respect to the gradients and the second moment (uncentered variance), which is required for learning rates accommodated for each parameter. The Adam update rules are governed by:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned} \tag{4.23}$$

where:

- m_t and v_t are estimates of the first and second moments respectively,
- β_1 and β_2 are the decay rates of these moments,
- $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected estimates,
- η is the learning rate,
- ϵ is a small constant to avoid division by zero.

Adam is known for its ability to handle sparse gradients and its fast convergence properties [33].

4.3.5. Adamax

Adamax is actually another form of decomposition of the Adam technique into the infinity norm of the gradients. This allows for a more robust estimate when large gradients are observed. The update rules for Adamax are:

$$u_t = \max(\beta_2 u_{t-1}, |g_t|), \quad \theta_{t+1} = \theta_t - \frac{\eta}{u_t} m_t \tag{4.24}$$

where:

- u_t gives the maximum observed metadensity up to time t,
- $g_t = \nabla_\theta L(\theta_t)$ is the gradient of the loss function,
- β_2 is the decay rate,
- η is the learning rate,
- m_t is the first moment estimate.

Adamax is particularly effective when dealing with high-dimensional spaces [33].

4.3.6. Nadam (Nesterov-accelerated Adaptive Moment Estimation)

Nadam blends in Adam's Adaptive Gradient Descent with Nesterov momentum from regularized terms. The update rules for Nadam are:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_t + \frac{(1-\beta_1)}{1-\beta_1^t} g_t \right) \end{aligned} \tag{4.25}$$

where:

- m_t and v_t are estimates of the first and second moments respectively,
- β_1 and β_2 are the decay rates,
- η is the learning rate,
- $\hat{v}_t$ is the bias-corrected estimate of the second moment,
- ϵ is a small constant to avoid division by zero.

Nadam incorporates the Nesterov momentum adaptation for faster convergence and improved performance [34].

4.4. Pipeline

The custom data processing pipeline handled the time-series data that were drawn from the database. The data were then broken down into fixed-length sequences, and the individual segments were used as input to the model. Finally, an important parameter for segment length was the segment size, which determined the temporal context that the model had at its disposal

in activity recognition.

The dataset was broken into training data, validation data, and test data such that it separated the sessions with respect to the subjects, performed to avoid any data leakage, where 80% is used for training, 10% is used for validation, and 10% is used for testing. This means that the model will be tested on completely unseen subjects. This will then allow an 8:1:1 split of the volume of recordings by eight different subjects into a training, validation, and test dataset. Justified by dataset diversity, substantial volume, and potential to yield more relevant kinds of samples caused by shifts of windows, growing value, and variety.

Dynamic data augmentation was used for model learning from the data. The shifting mechanism of dynamic data augmentation creates numerous overlapping segments from each session, which in effect increases the number of training examples.

The model was prepared using TensorFlow and Keras for making flexible architectures, such as taking into account the different types of convolutional and recurrent layers to properly extract features. Other principal hyperparameters present in the model preparation include the segment size, batch size, learning rate, and finally, choosing the optimizer.

Model performance was evaluated by guiding the hyperparameters from fitting the training data and changes in the model by the validation data. Testing of the developed model was finally performed with the test dataset for evaluating the above metrics, and summarizing the results of the study, proving or disproving model's generalization.

4.5. Hyperparameter Optimization

The aim of this study has been to tune the hyperparameters for good performance in classification tasks embedded in the MHEALTH datasets. A systematic search for hyperparameters was performed to provide improved performance on the validation set.

The hyperparameters considered for the study include:

- ***segment size*** that represent the actual length of a time series taken as input to the model, different sizes for this segment were experimented with in order to determine the ideal length for capturing activity patterns;
- ***batch size*** that refers to the number of segments processed in a single iteration of training;
- ***number of shifts*** which defines the number of shifts and new starting points for segment creation in each pass;
- ***learning rate*** calculated at each epoch determines the step taken toward minimizing the loss function;
- ***number of convolutional layers*** which defines the model's depth: the number of convo-

lutional layers;

- ***number of recurrent layers*** which defines the model's depth: the number of recurrent layers;
- ***number of dense layers*** which defines the model's depth: the number of dense layers;
- ***units*** explained by the number of filters in convolutional layers and units in the recurrent and dense layers, different combinations where tested;
- ***dropout rates*** that randomly drop a fraction of neurons during training, this served as a model regularization procedure, improving generalization;
- ***optimizer types*** were tested to determine the most appropriate algorithm for the dataset and model architecture;
- ***varius convolutional layer configurations*** were tested, filter sizes, and kernel sizes, additionally, the inclusion of batch normalization was tested to assess its impact on training stability and model performance;
- ***pooling type and size***, pooling strategies were employed, max pooling and average pooling.

Hyperparameters were optimized using the framework Optuna [14]. Reevaluated against the validation set. The goal was to achieve the lowest validation loss and highest validation accuracy, thus finding an optimal set of hyperparameters. The best settings were identified and tested against a validation set. The model was tested with different sets of hyperparameters. Each model employed an early stopping mechanism, and the weights achieved at peak performance were restored. This selection process ensured that the model's performance was protected against overfitting, with the best attributes saved. The best-performing model on the validation data was selected, and its performance was checked against the test set to prove the generalization achieved by the best-performing model.

4.6. Explored Samplers

This study investigated the effectiveness of five different, heterogeneous sampling methods described in this section.

4.6.1. Random Search

One of the simplest and most common methods to optimize hyperparameters is random search. This method involves randomly sampling hyperparameters θ from the defined search spaces:

$$\theta_i \sim \mathcal{U}(a_i, b_i) \tag{4.26}$$

where:

- θ_i is the i-th hyperparameter,
- $\mathcal{U}(a_i, b_i)$ represents the uniform distribution within the limits a_i and b_i for the i-th hyperparameter,
- a_i and b_i are the lower and upper bounds of the search space for the i-th hyperparameter.

Random Search is simple but can be effective in high-dimensional spaces [7].

4.6.2. Quasi-Monte Carlo (QMC)

Quasi-Monte Carlo methods improve random search algorithms by using low-discrepancy sequences, which enable a more even sampling of hyperparameters within the search space. The aim is to reduce the discrepancy, D_n, for the sequence:

$$D_n = \sup_{B \subseteq [0,1]^s} \left| \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_B(\mathbf{u}_i) - \lambda(B) \right| \tag{4.27}$$

where:

- $\mathbf{u}_i$ are points in the s-dimensional unit cube,
- $\lambda(B)$ is the Lebesgue measure of the set B.

This method provides a more equitable representation of the hyperparameter space [36].

4.6.3. Tree-structured Parzen Estimator (TPE)

The Tree-structured Parzen Estimator (TPE) is a method in Bayesian optimization where the objective function $f(\theta)$ is characterized by a probabilistic framework through two distinct densities:

$$l(\theta) = P(\theta | f(\theta) < f^*) \quad \text{and} \quad g(\theta) = P(\theta | f(\theta) \geq f^*) \tag{4.28}$$

where:

- f^* is a threshold across which the observed function values are bimodally scattered.

The TPE algorithm seeks hyperparameters by maximizing the ratio $l(\theta)/g(\theta)$, where this ratio measures where improvements are relatively more likely inside the parameter space [13].

4.6.4. Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

The CMA-ES is an iterative mover of the multivariate normal distribution, with the covariance adjusted to allow fine sampling in the most promising regions of the search space:

$$\begin{aligned} \mathbf{m}_{t+1} &= \mathbf{m}_t + c_m \mathbf{p}_t \\ \mathbf{C}_{t+1} &= (1 - c_c)\mathbf{C}_t + c_c \mathbf{p}_t \mathbf{p}_t^T \\ &\quad + c_\mu \sum_{i=1}^{\mu} w_i (\mathbf{x}_i - \mathbf{m}_t)(\mathbf{x}_i - \mathbf{m}_t)^T \end{aligned} \tag{4.29}$$

where:

- $\mathbf{m}_t$ is the mean vector at generation t,
- c_m, c_c, and c_μ are learning rates,
- $\mathbf{p}_t$ is the evolution path,
- $\mathbf{C}_t$ is the covariance matrix at generation t,
- w_i are the weights assigned to the offspring.

CMA-ES is particularly effective in complex, multimodal optimization landscapes [37].

4.6.5. Bayesian Optimization with Gaussian Process (GP)

Gaussian Process Regression is a Bayesian Optimization technique that characterizes the objective function through a Gaussian Process framework:

$$f(\theta) \sim \mathcal{GP}(\mu(\theta), k(\theta, \theta')) \tag{4.30}$$

where:

- $\mu(\theta)$ is the mean function,
- $k(\theta, \theta')$ is the covariance function.

The acquisition function $\alpha(\theta)$ is used to balance exploration and exploitation, guiding the choice of the next point θ_{t+1} to make an observation at:

$$\theta_{t+1} = \arg\max_{\theta} \alpha(\theta|\mathcal{D}_{1:t}) \tag{4.31}$$

where:

- $\mathcal{D}_{1:t}$ is the data observed up to iteration t.

This approach is especially powerful for costly-to-evaluate functions, as Bayesian Optimization with GP aims to find a global optimum with relatively few evaluations [38].

4.7. Search and optimization of the target model

The hyperparameter study was arranged into a four-step program to optimize the model in a systematic manner. To begin with, a baseline model was created as a means to ensure an optimal comparison for future in-depth optimizations. An initial hyperparameter search was performed with different strategies and parameters to determine the input size, a key parameter that defines the future search spaces, as well as the maximum value for the number of shifts that would be applied in all subsequent fine-tuning experiments. In step 3, the model architecture was optimized. The best architecture was selected, including the number of recurrent layers; inclusion, number of recurrent and dense layers. The last step involved tuning the finalized architecture, dropout rates, numbers of units and additional configurations for already defined layer composition. Figure 4.1 shows the proposed four-step study plan, which facilitated a thorough exploration and utilization of all capabilities of the custom tuning setup established to provide an optimal solution to this classification task.

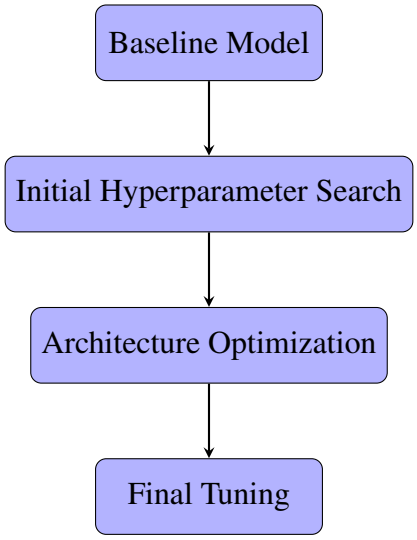

Figure 4.1: Visualization of the four-step hyperparameter study plan.

5. Hyperparameter Study

5.1. Baseline Model

This section describes the baseline model used for comparison in this study, providing details on its architecture, training process, and evaluation results.

5.1.1. Model Architecture

The basic model architecture is shown in Figure 5.1 consists of a GRU layer with 128 units, which is followed by a dense output layer. GRU layer captures the temporal aspects of the input sequences. The final dense layer has 12 units, one for each activity class, connected with a softmax activation function to provide class probabilities. The model was trained using sequences of 50 samples, hence the input layer is `[50,23]`.

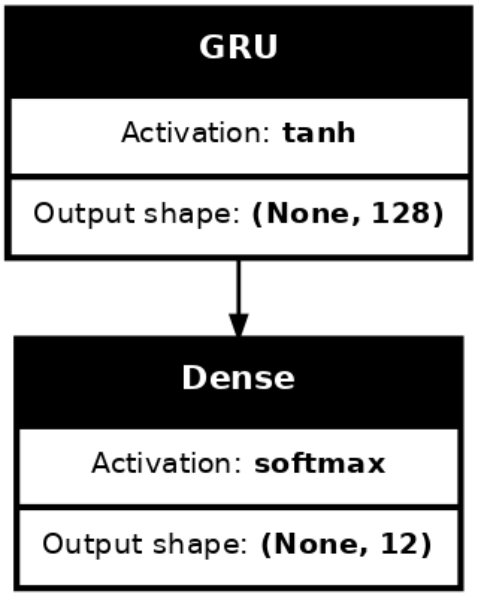

Figure 5.1: Architecture of the Baseline Model

The decision to use a recurrent neural network architecture, specifically a Gated Recurrent Unit, was influenced by the nature of the data, which involves sequential time-series information. RNNs and, more particularly, GRUs are naturally best-fit architectures for capturing temporal

pattern and dependencies in such tasks. Therefore, it would be good in fitting this task that belongs to sequence understanding.

Since GRUs are lighter structured and more efficient in computation than LSTM units, they were used instead of LSTMs. Since these models have fewer hyperparameters, they are easier to implement and tune; these are the reasons for considering them is in this study. The use of a single recurrent layer with GRUs allows a relatively simple comparison against the more complex models that are subsequently delved into with this thesis. The GRU-based architecture was preferred because it achieves a good tradeoff between performance and computational efficiency, representing a suitable candidate to set a strong baseline in this work.

5.1.2. Training Process

The baseline model was trained using the Adam optimizer, with a learning rate of 0.001, a batch size of 100. Model was trained without applying shifts to the data. An early stopping mechanism was used to prevent overfitting and ensure the model's generalization. This approach halted training if no improvement in validation loss was observed, ensuring that the model retained the best weights and did not undergo unnecessary training.

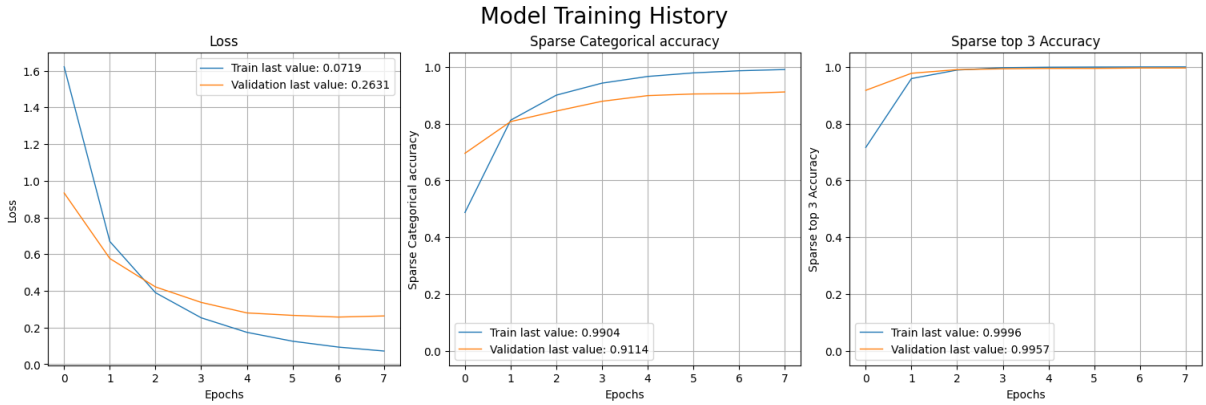

Figure 5.2: Baseline Model's Training History

The training history in Figure 5.2 shows a consistent decrease in training loss, demonstrating effective learning, while the validation accuracy remained robust, indicating good generalization.

5.1.3. Evaluation on the Test Set

The model was evaluated on a separate test set, with the results shown in Figure 5.3. The model achieved a test loss of 0.1511, a sparse categorical accuracy of 93.71%, and a sparse top-3 accuracy of 100%. These results show strong performance for a baseline model.

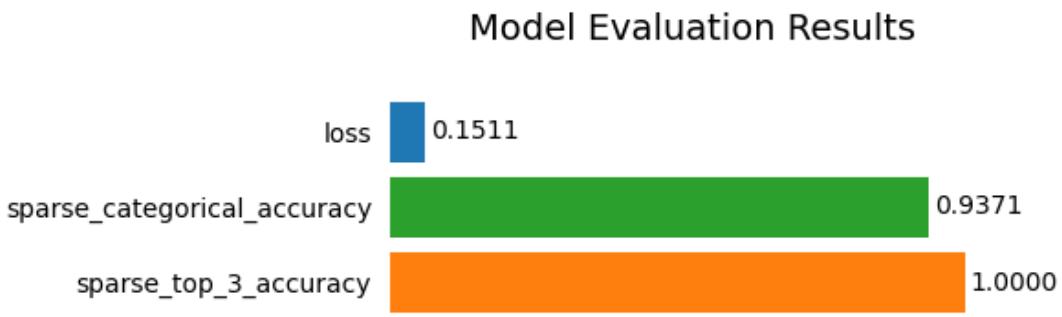

Figure 5.3: Baseline Model's Evaluation on the Test Set

The baseline model, despite its lack of complexity, performed well across all key metrics, making it a good reference for comparisons.

5.2. Initial Search

The purpose of this section is to identify and establish the key parameters that define the search space for future tuning. This initial tuning task shares the architecture of the baseline model, with one recurrent GRU layer.

5.2.1. Hyperparameter Tuning Strategies

This search explores the initial hyperparameter tuning using the following strategies: Random Search, Quasi-Monte Carlo (QMC), Tree-structured Parzen Estimator (TPE), Covariance Matrix Adaptation Evolution Strategy (CMA-ES), and Bayesian optimization with Gaussian Process (GP).

The explored parameters are:

- `segment_size` that defines the size of the input segment and is chosen from the values 25, 50, 75, and 100, which translates to 0.5, 1, 1.5, and 2 seconds of sensor recordings that are passed to the input layer;
- `batch_size`, the batch size used during training selected from the range 25 to 250, with increments of 5;
- `n_shifts` that represents the number of shifts applied to the data (multiplication of train data by selecting multiple window shifts) and is chosen from the range 1 to 5;
- `l_rate`, the learning rate, selected from a logarithmic scale between 1×10^{-4} and 1×10^{-2}, controlling the adjustment of model weights.

Parameters `segment_size` and `n_shifts` not only affect the training process but also the usability and applicability of the model. The parameters `batch_size` and `l_rate` were included in this search to ensure that the best configuration of `segment_size` and `n_shifts` is properly trained and converges. This section reports the findings of the initial search and recommends the best parameters to be incorporated into the tuning of more complex models. It is crucial to determine this initial hyperparameter values before refining architectures, as this defines the scope of the entire study and ensures the viability of future optimizations based on this foundation.

5.2.2. Optimization History and Performance Analysis

The optimization history for each sampler is presented in Figures 5.4 through 5.8. These figures illustrate how each sampler explored the hyperparameter space and converged towards the best-performing models.

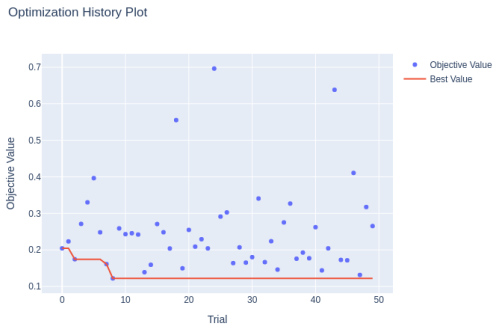

Figure 5.4: Initial Search, Optimization History Plot - Random Sampler

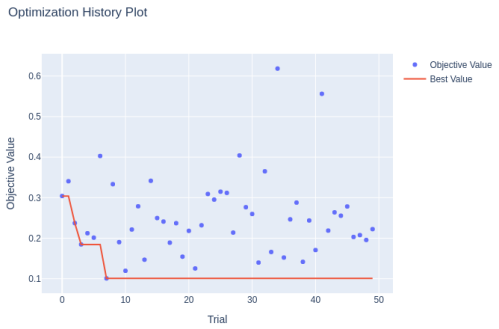

Figure 5.5: Initial Search, Optimization History Plot - QMC Sampler

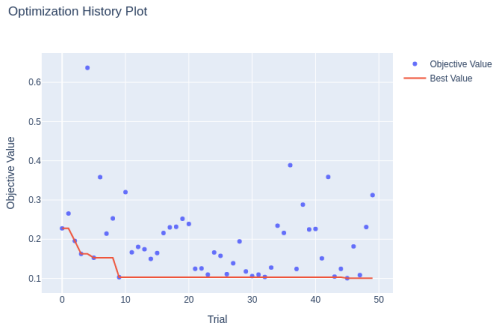

Figure 5.6: Initial Search, Optimization History Plot - TPE Sampler

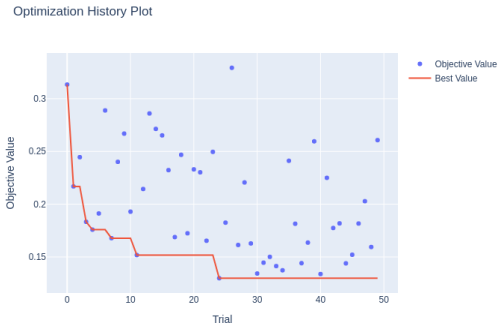

Figure 5.7: Initial Search, Optimization History Plot - CMA-ES Sampler

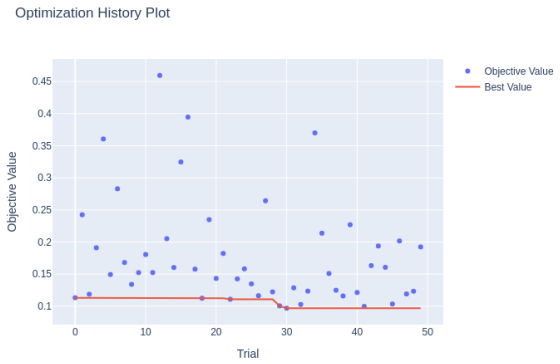

Figure 5.8: Initial Search, Optimization History Plot - GP Sampler

The results are summarized in the table 5.1. The GP Sampler is highlighted as it achieved the best validation loss. The findings from trials are reported:

Sampler	Random Sampler	QMC Sampler	TPE Sampler	CMA-ES Sampler	GP Sampler
Best Trial	8	7	45	24	30
Validation Loss	0.1220	0.1009	0.1006	0.1299	**0.0969**
Segment Size	50	75	50	50	50
Batch Size	25	50	35	160	100
Number of Shifts	3	5	4	3	3
Learning Rate	0.0021	0.0018	0.0013	0.0027	0.0033

Table 5.1: Initial Search, Summary of the best trials and optimal hyperparameters for different samplers

5.2.3. Hyperparameter Importance and Impact

Figures 5.9 through 5.13 illustrate the hyperparameter importances for different sampling techniques. These visualizations highlight which hyperparameters most significantly impacted model performance during the optimization process.

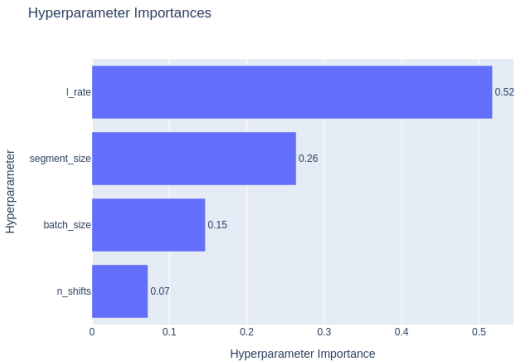

Figure 5.9: Initial Search, Hyperparameter Importances - Random Sampler

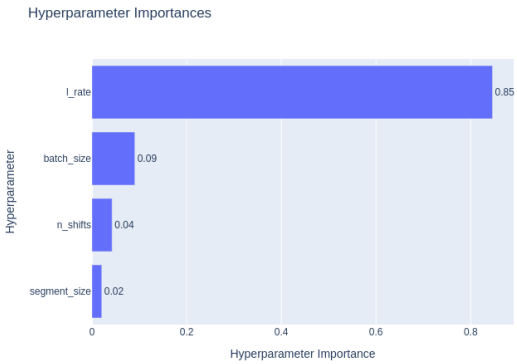

Figure 5.10: Initial Search, Hyperparameter Importances - QMC Sampler

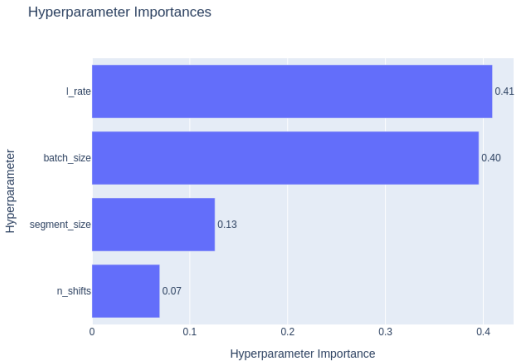

Figure 5.11: Initial Search, Hyperparameter Importances - TPE Sampler

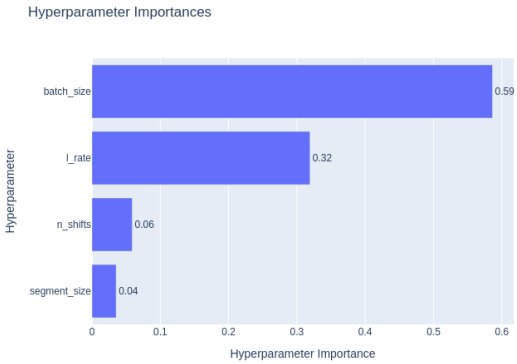

Figure 5.12: Initial Search, Hyperparameter Importances - CMA-ES Sampler

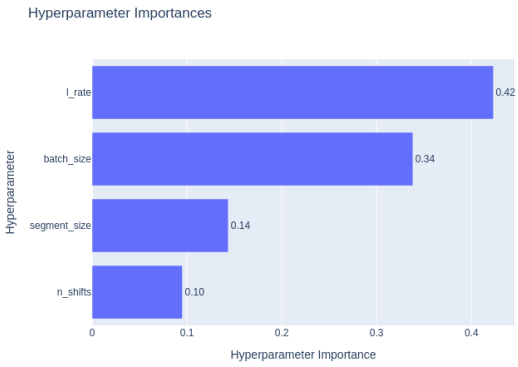

Figure 5.13: Initial Search, Hyperparameter Importances - GP Sampler

Samplers prioritize hyperparameters differently. The learning rate, `l_rate`, is consistently significant across most samplers, with the QMC and TPE samplers showing it as the most

critical factor. However, the importance of other parameters varies: `batch_size` plays a dominant role in the CMA-ES sampler, while `segment_size` is more influential in the TPE and Random samplers. Different optimization strategies rely on different hyperparameter dynamics to achieve optimal performance.

The parallel coordinate plots (Figures 5.14 through 5.18) provide more inside, showcasing the relationships between hyperparameters and model performance.

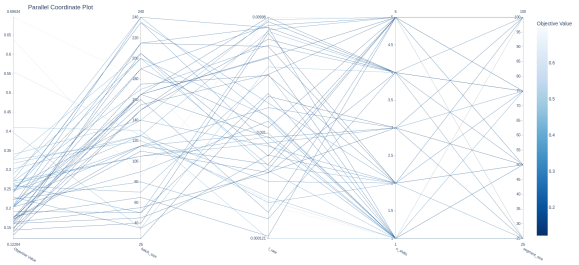

Figure 5.14: Initial Search, Parallel Coordinate Plot - Random Search

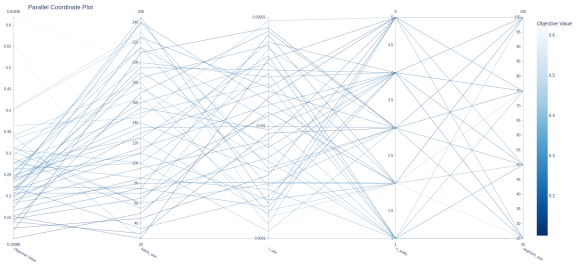

Figure 5.15: Initial Search, Parallel Coordinate Plot - QMC Sampler

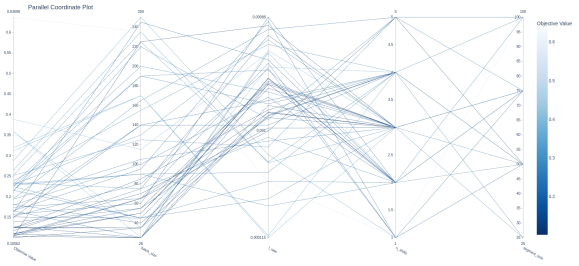

Figure 5.16: Initial Search, Parallel Coordinate Plot - TPE Sampler

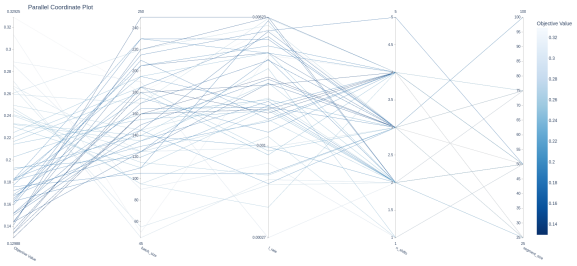

Figure 5.17: Initial Search, Parallel Coordinate Plot - CMA-ES Sampler

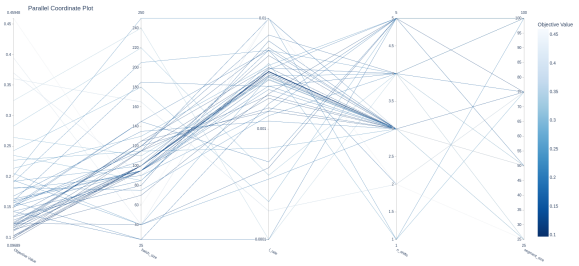

Figure 5.18: Initial Search, Parallel Coordinate Plot - GP Sampler

Figures 5.19 through 5.23 illustrate the slice plots for the best model training histories of all studied samplers.

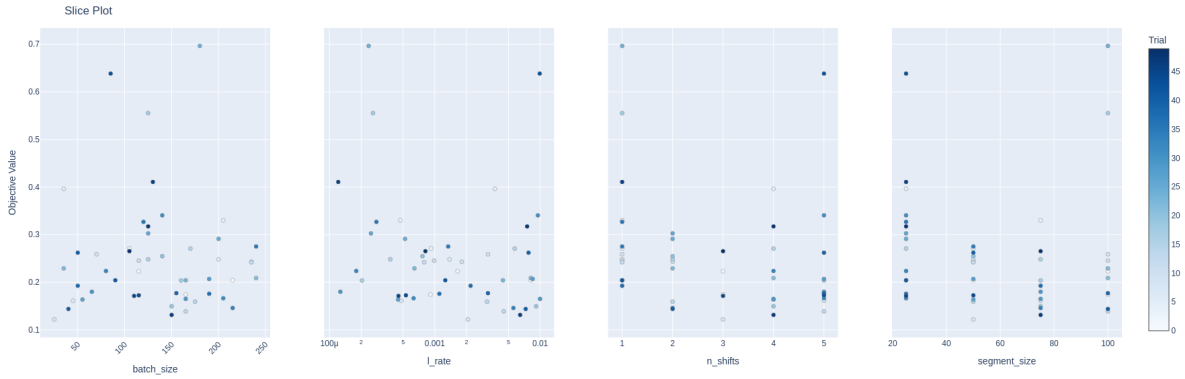

Figure 5.19: Initial Search, Slice Plot - Random Sampler

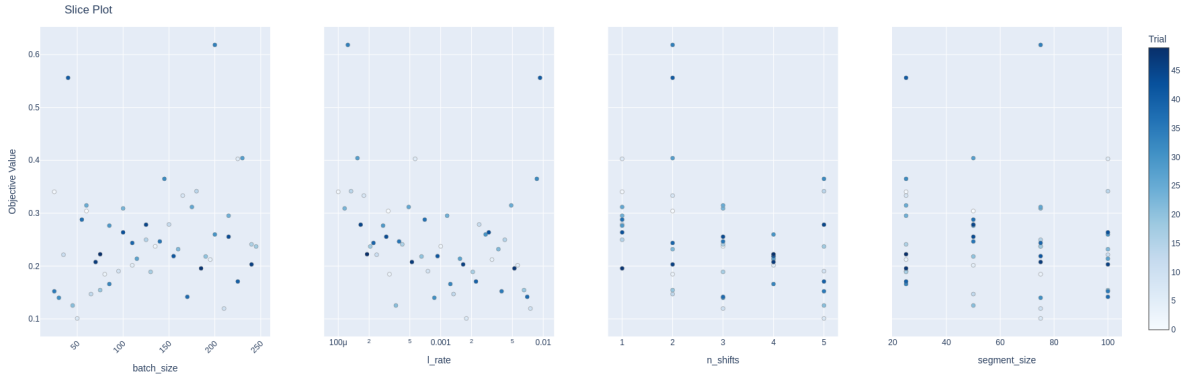

Figure 5.20: Initial Search, Slice Plot - QMC Sampler

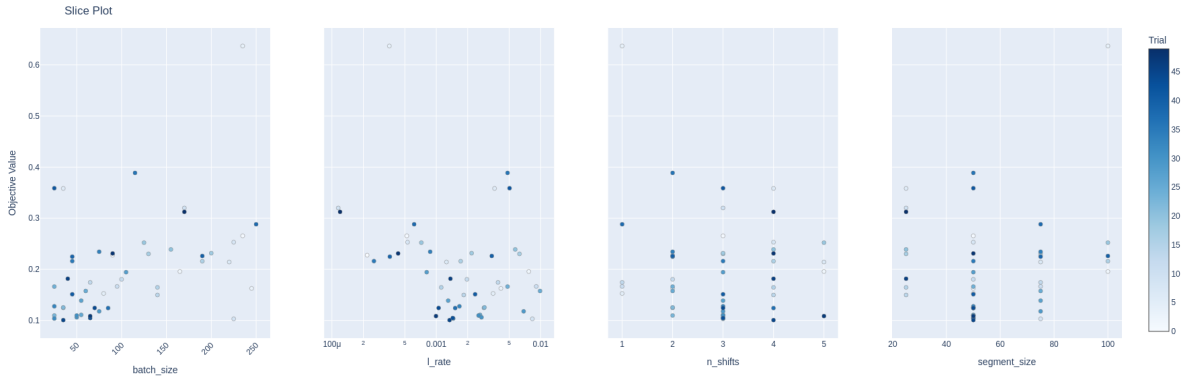

Figure 5.21: Initial Search, Slice Plot - TPE Sampler

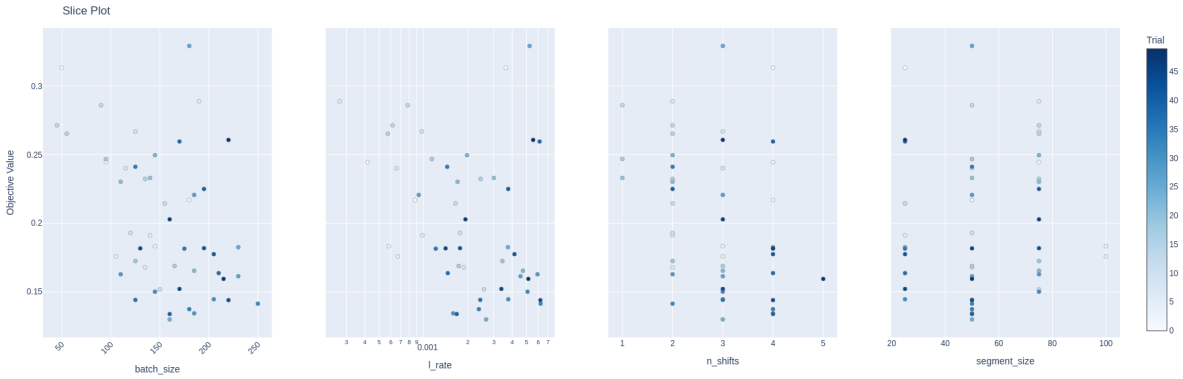

Figure 5.22: Initial Search, Slice Plot - CMA-ES Sampler

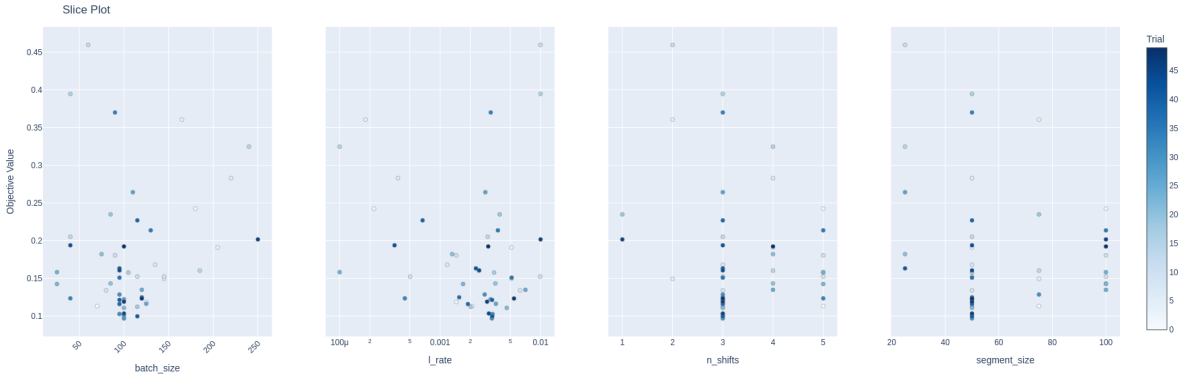

Figure 5.23: Initial Search, Slice Plot - GP Sampler

Using 4 shifts and a segment size of 50 generally gives improved results, but 3 shifts often show the lowest validation loss in the slices. While a segment size of 75 sometimes performs well, 50 is more reliable and practical as it matches a 1-second data window at 50 Hz, making it more appropriete for real-time sequence classification. Lowering the segment size would also create a more efficient model that requires less data for classification and adapts to new activities more quickly. Batch sizes are varied and do not show a clear pattern, so they may need to be tuned for each specific model, nevertheless, lower batch seem to perform better, and lower values were studied in future tunings. Learning rates around 0.001 to 0.002, with values closer to 0.002 often leading to better performance.

5.2.4. Optimized Models Evaluation

Figures 5.24 through 5.28 provides insight into how the models trained with the optimal hyperparameters. All models employed early stopping.

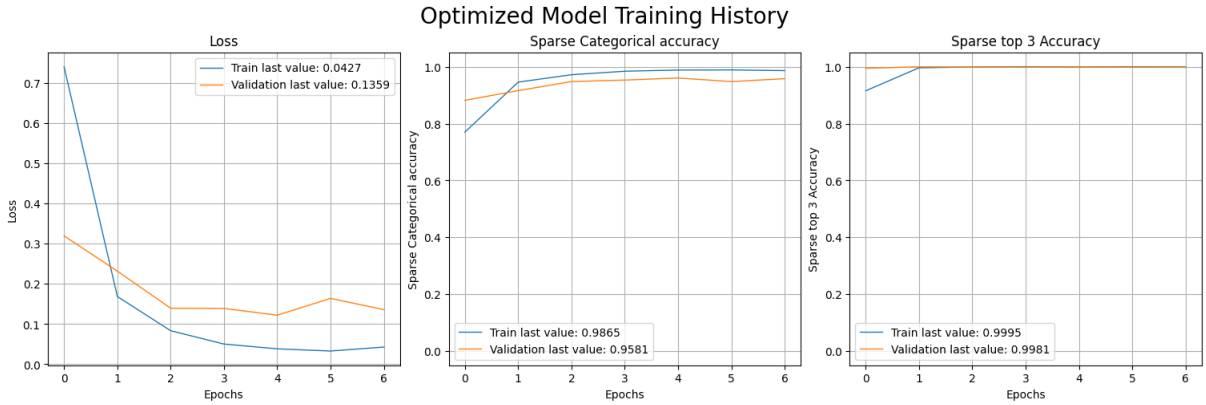

Figure 5.24: Initial Search, Best Model Training History - Random Sampler

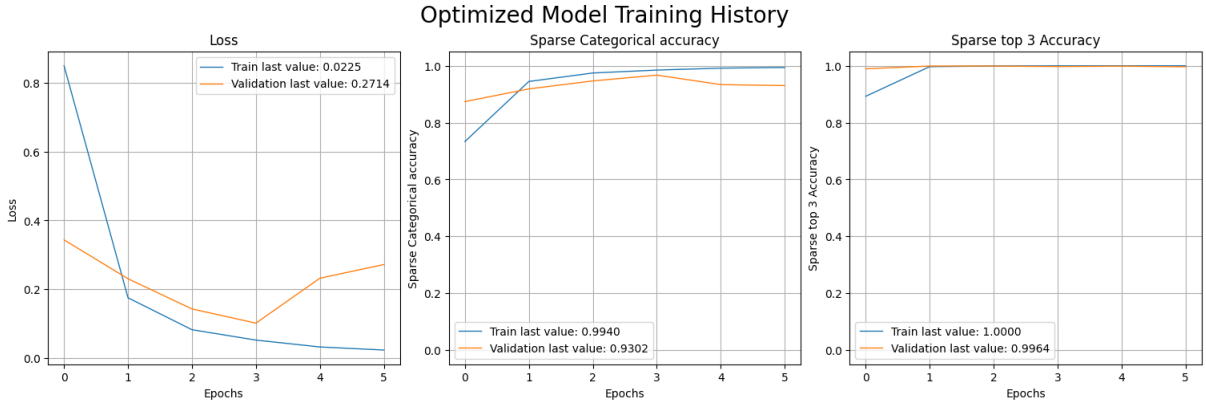

Figure 5.25: Initial Search, Best Model Training History - QMC Sampler

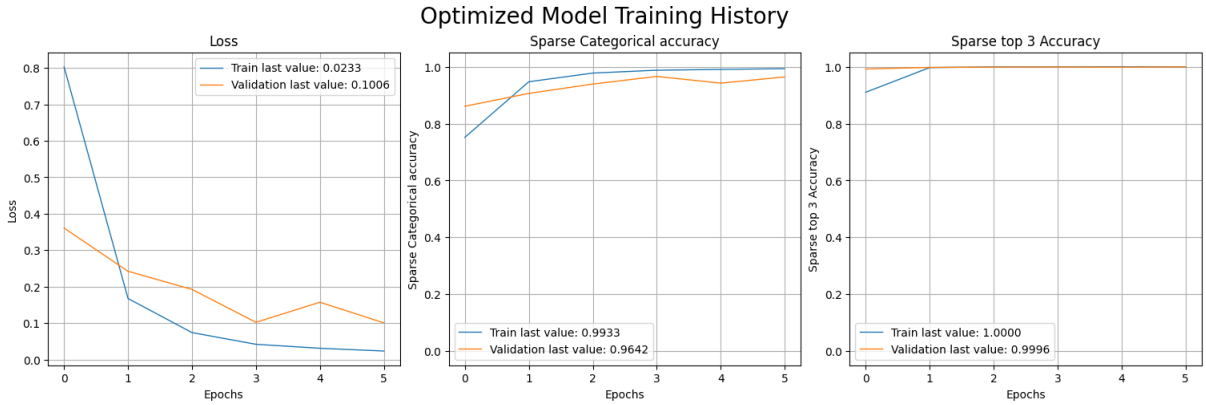

Figure 5.26: Initial Search, Best Model Training History - TPE Sampler

Figure 5.27: Initial Search, Best Model Training History - CMA-ES Sampler

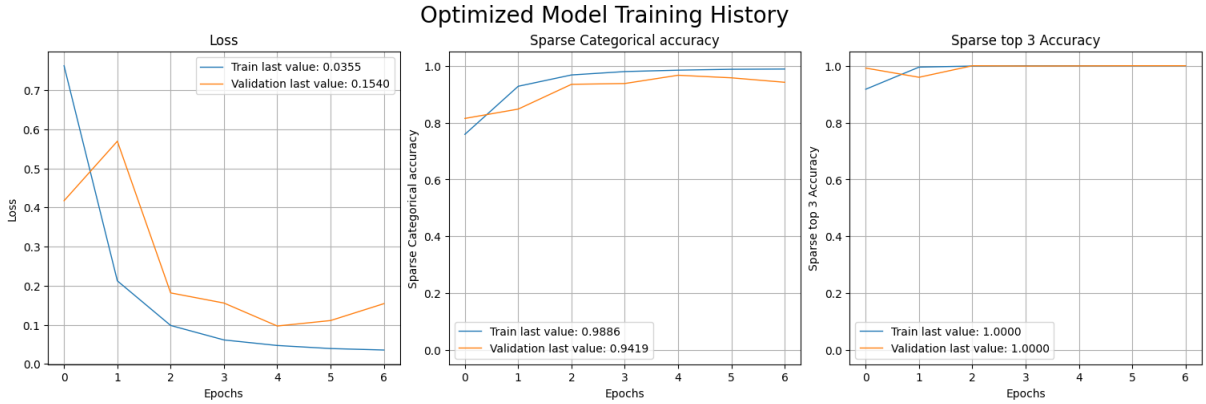

Figure 5.28: Initial Search, Best Model Training History - GP Sampler

GP Sampler achieved the lowest validation loss of 0.0969, but the history plot indicates overfitting due to a small gap between training and validation accuracy. Other Samplers, Random, QMC, CMA-ES, while performing decently, had higher validation losses and slightly less consistent accuracy, did not generalize as well as TPE or GP. The TPE Sampler provided the best balance between accuracy and generalization, making it the most effective among the models. The TPE Sampler identified 4 shifts and a segment size of 50 as optimal parameters, which also provided strong results across trials with all samplers. While batch size and learning rate varied, further refinement is necessary and can be explored with different architectures.

5.3. Architecture Search

The purpose of this section is to investigate and optimize the neural network architecture of the target model. This architecture search is an exhaustive investigation of different hyperparameters that configure the network. By adjusting these parameters, the best configuration is defined.

5.3.1. Hyperparameters defining architecture

Number of shifts and segment size are fixed at 4 and 50, respectively. The following parameters are explored in this architecture search:

- `batch_size` that defines the number of samples processed in one training iteration, selected from the range of 25 to 50, with increments of 5;
- `optimizer`, the optimizer is chosen from Adam, SGD, RMSprop, Adamax, and Nadam, described in Section 4.3;
- `l_rate`, the learning rate, selected from a logarithmic scale between 1×10^{-3} and 1×10^{-2}, controlling the adjustment of model weights;
- `n_conv_layers` defining the number of convolutional layers, selected from 0 to 3, allowing the model to extract hierarchical features from the input data;
- `conv_filters` that defines the number of filters, chosen from 16 to 128, with increments of 16, determining the complexity of the feature extraction;
- `recurrent_type`, the recurrent layer type is chosen between GRU and LSTM;
- `n_recurrent_layers`, the number of recurrent layers is selected from 1 to 3, which influences the depth of temporal feature processing;
- `recurrent_units`, the number of units in each recurrent layer is selected from 32 to 256, with increments of 16, defining the model's capacity to capture temporal dependencies;
- `n_dense_layers`, the number of dense layers, selected from 0 to 2, used for combining features;
- `dense_units`, the number of units in each dense layer, selected from 32 to 128, with increments of 16, affects the ability to learn complex relationships;
- `use_batch_norm`: Specifies whether to apply batch normalization, which can improve training speed and stability.
- `pooling_type`: The type of pooling operation is selected between max pooling and average pooling, which reduces the dimensionality of the data while retaining important features.
- `pool_after_each`: Specifies whether to apply pooling after each convolutional layer or only at the last one.

Given the categorized nature of the parameters, Random Search and Tree-structured Parzen Estimator (TPE) were employed as the sampling strategies for the architecture search. Though QMC, CMA-ES, and Bayesian optimization with GP have been explored in previous works, they are not included in the current one due to their applicability in continuous search spaces,

instead of categorized parameters. A random search was executed as a baseline approach with 50 trials to possibly cover most architecture configurations. On the other hand, as TPE is more efficient for optimization in categorized and mixed search spaces, 100 trials were performed to identify the best parameters.

5.3.2. Tuning Results

The optimization history for the Random and TPE is presented in Figures 5.29 and 5.30.

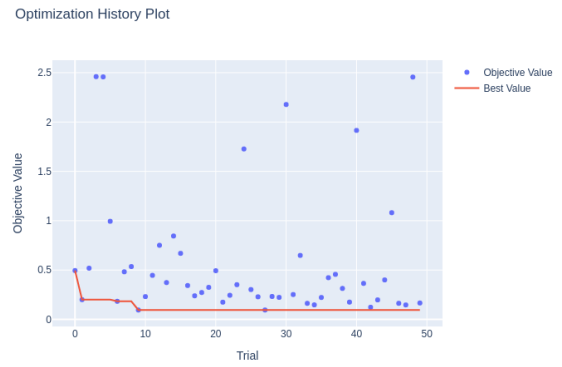

Figure 5.29: Architecture Search, Optimization History - Random Sampler

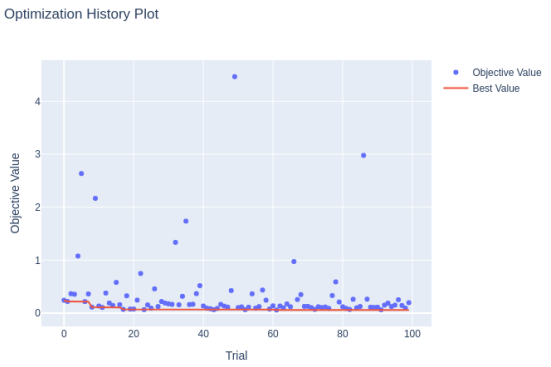

Figure 5.30: Architecture Search, Optimization History - TPE Sampler

In the Random Search trials, the best-performing model achieved a value of 0.095 with a batch size of 50, using the Adamax optimizer with a learning rate of 0.0026. This model was configured with a GRU architecture featuring 2 recurrent layers (128 units in the first and 48 units in the second) and 3 convolutional layers, employing max pooling without batch normalization. Another notable configuration achieved a value of 0.183 with a batch size of 30, utilizing GRU layers with batch normalization.

The TPE method identified the most effective configuration, achieving a value of 0.059 with a batch size of 25, using the Adam optimizer with a learning rate of 0.0013. This model featured a single GRU layer with 208 units and one convolutional layer with 32 filters, also using max pooling without batch normalization. Other strong configurations with values of 0.065 and 0.066 were observed using similar setups with batch sizes of 35 and 40.

LSTM networks did not establish themselves as top performers; both sampler studies had chosen GRU layers over LSTM. Additionally, models that had more dense layers after recurrent did not perform well.

Hyperparameter importances are presented in Figures 5.31 and 5.32.

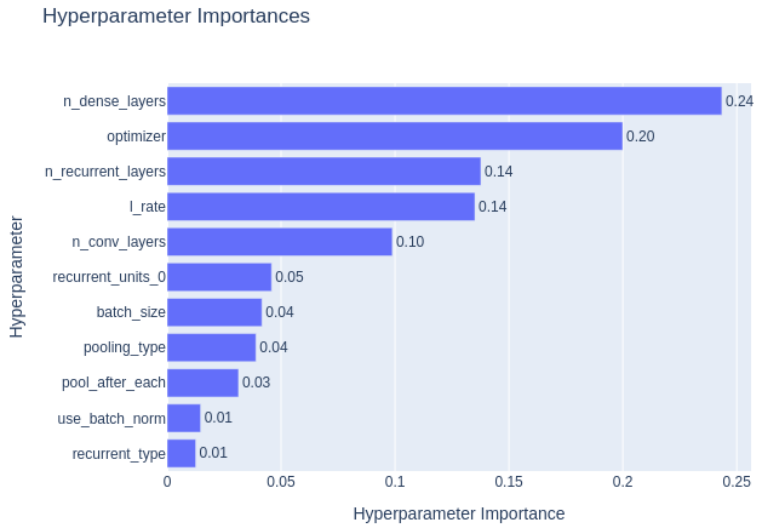

Figure 5.31: Architecture Search, Hyperparameter Importances - Random Sampler

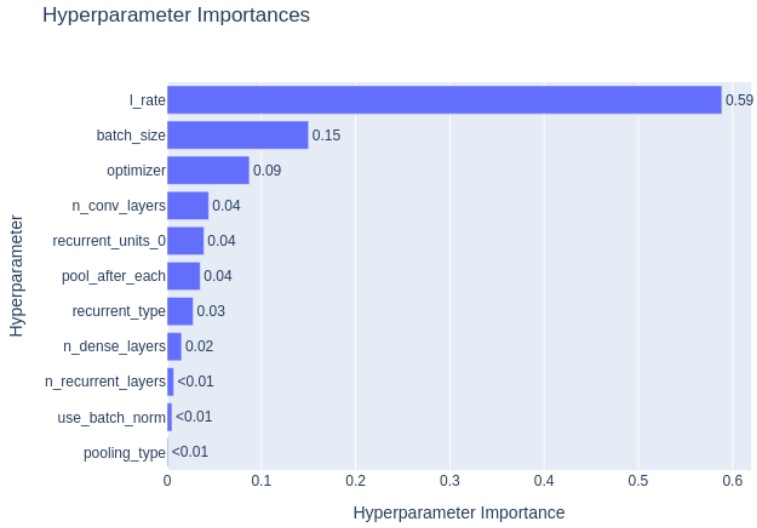

Figure 5.32: Architecture Search, Hyperparameter Importances - TPE Sampler

The smaller batch size was helpful in finding better performing model, but that was not the main factor. More importantly, the type of optimizer used, Adam or Adamax, a learning rate of about 0.001 to 0.002, and simplicity in model architecture: fewer convolutional and recurrent layers, led to better performance. TPE outperformed Random Search in finding better configurations, which shows the added value of using more refined optimization method.

The tuning resulted in the models visualized in Figures 5.33 and 5.34.

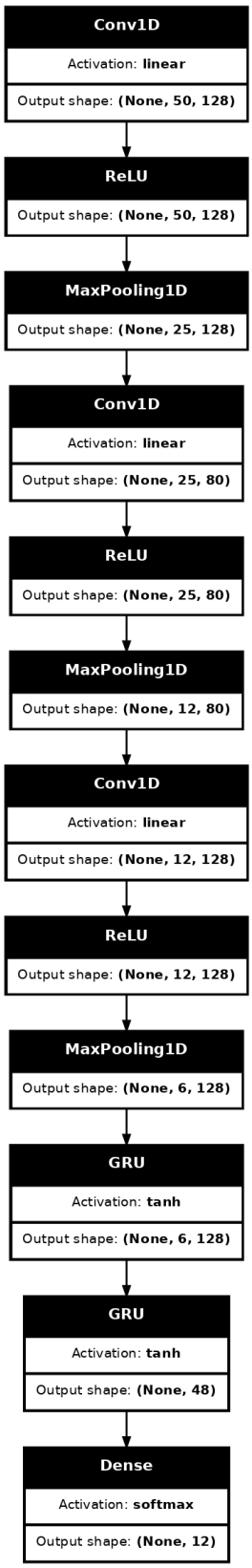

Figure 5.33: Architecture Search, Produced Model - Random Sampler

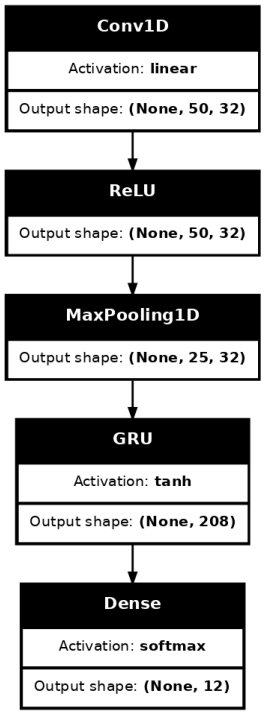

Figure 5.34: Architecture Search, Produced Model - TPE Sampler

Figures 5.33 illustrates the architecture derived from the Random Search method. This model is characterized by a series of three Conv1D layers, each followed by ReLU activation and MaxPooling1D layers, leading to a deep feature extraction process. The model then transitions into two GRU layers, which are responsible for capturing sequential dependencies in the data.

Figures 5.34 shows the architecture selected by the TPE method, which is notably simpler compared to the Random Search model. This model begins with a single Conv1D layer followed by ReLU activation and MaxPooling1D. It then feeds into a single GRU layer with 208 units.

Figures 5.35 and 5.36 present histories of training of the best performing models.

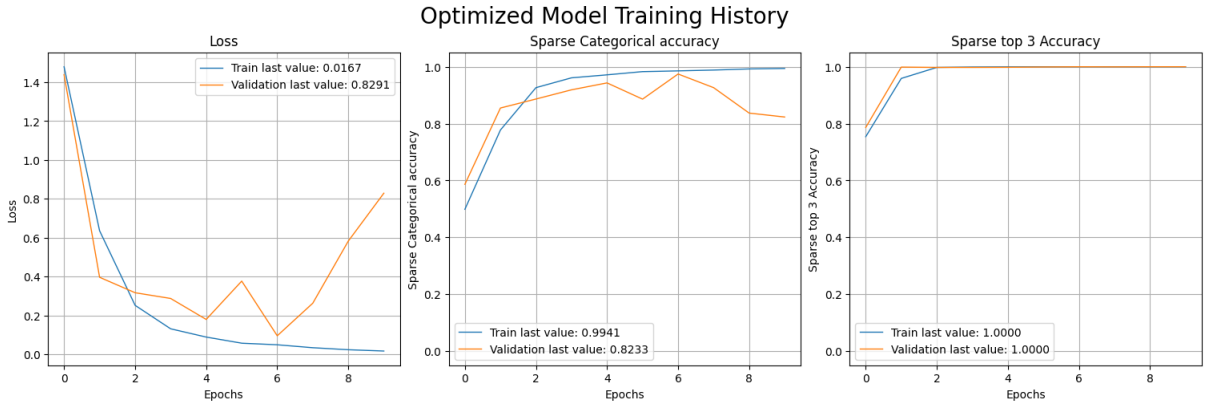

Figure 5.35: Architecture Search, Training History - Random Sampler

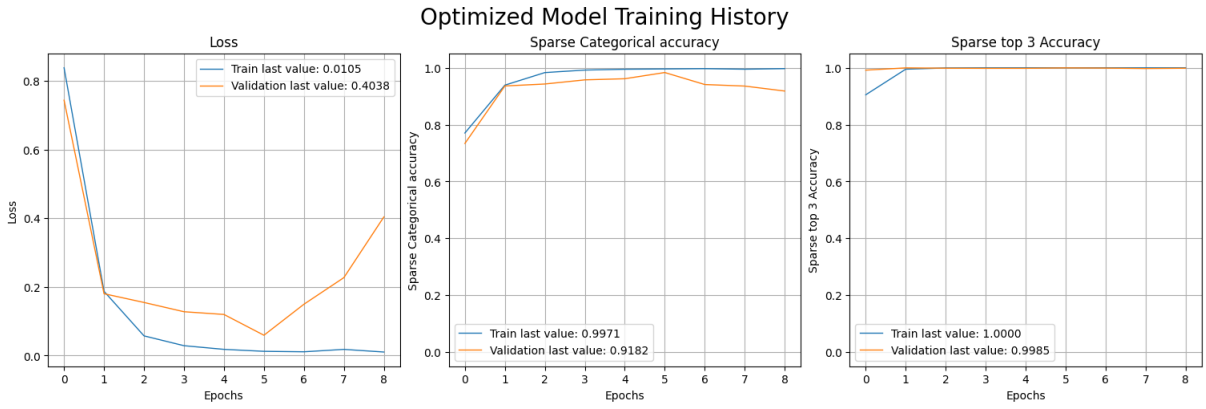

Figure 5.36: Architecture Search, Training History - TPE Sampler

The TPE Sampler model outperforms the Random Sampler model in terms of validation accuracy and minimal loss. The TPE model demonstrates better generalization, as evidenced by higher validation accuracy and lower validation loss. The model proposed by the TPE Sampler, which features one recurrent layer following the convolutional layers, will be used for the final tuning of the target model. Both optimized models, at their peak performance with restored weights, outperformed the models proposed in the initial search study.

5.4. Final Tuning

This section presents the results of final tuning with the architecture defined in the previous section.

5.5. Final Tuning parameter space

The number of shifts and batch size parameter are fixed. Additionally, the core architecture choices like number of convolutional and recurrent layers are fixed at 1, `pooling_type` is fixed as max pooling. GRU was chosen for recurrent layer types, `recurrent_type`. Batch normalization is not applied in this architecture as it did not provide improvements in the architecture search. Adam with its best performance in previous tuning is chosen as the `optimizer`.

This search study includes new hyperparameters: `dropout_rates`, `kernel_size` and `pool_size` that were not included in earlier tunings, and where added to fully explore the model's capabilities. In previous searches smaller batch sizes performed better, consequently lower values for batch sizes where tested.

The following parameters are explored in this target model tuning:

- `batch_size` defines the number of samples processed in one training iteration, selected from a range of 10 to 50;
- `l_rate` selected from a logarithmic scale between 1×10^{-4} and 1×10^{-2}, controlling the adjustment of model weights;
- `recurrent_units`, defining the model's capacity to capture temporal dependencies, the number of units in each recurrent layer is selected from 32 to 512;
- `dropout_rates` help prevent overfitting by randomly dropping units during training, for each layer selected from 0.0 to 0.5;
- `conv_filters` is the number of filters selected from 16 to 128, determining the complexity of the feature extraction;
- `kernel_size` determines the size of the convolutional kernel, which affects the receptive field of the convolutional operation, selected from 3 to 7;
- `pool_size` controls the degree of downsampling applied to the data, selected from 2 to 5.

5.5.1. Optimization History

The optimization history for each sampler is presented in Figures 5.37 through 5.41, these illustrate how each sampler explored the hyperparameter space and converged towards the best-performing models.

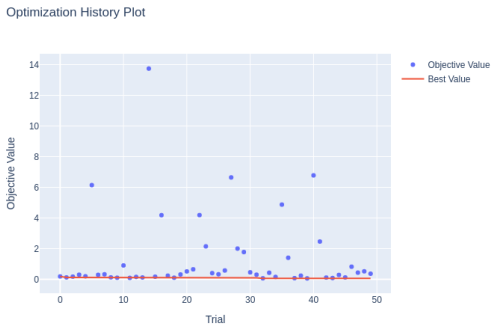

Figure 5.37: Final Tuning, Optimization History Plot - Random Sampler

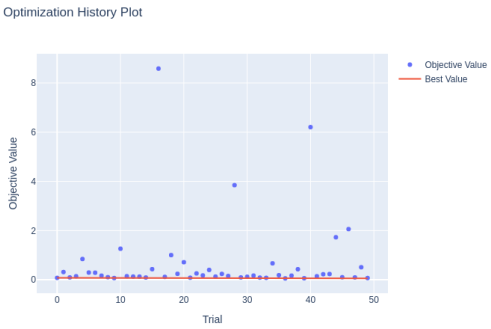

Figure 5.38: Final Tuning, Optimization History Plot - QMC Sampler

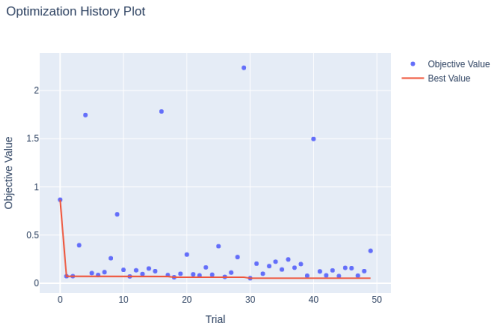

Figure 5.39: Final Tuning, Optimization History Plot - TPE Sampler

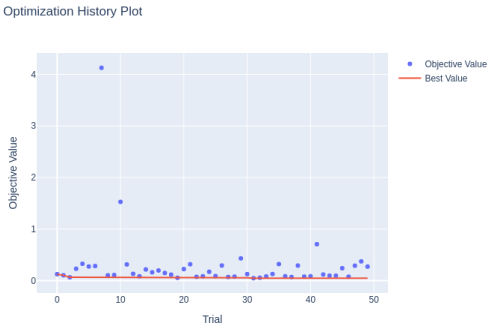

Figure 5.40: Final Tuning, Optimization History Plot - CMA-ES Sampler

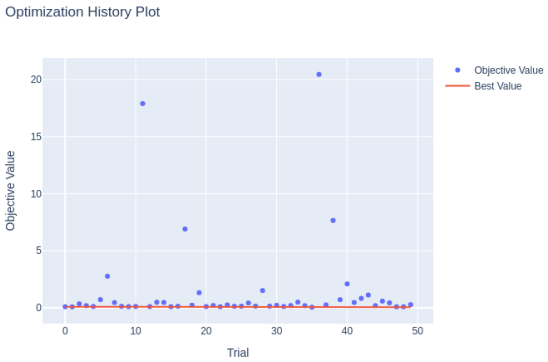

Figure 5.41: Final Tuning, Optimization History Plot - GP Sampler

The search for the best parmaeters could be summarized by table 5.2. CMA-ES sampler column is marked since this sampler produced the model with the lowest validation loss.

Sampler	Random Sampler	QMC Sampler	TPE Sampler	CMA-ES Sampler	GP Sampler
Best Trial	39	36	30	31	35
Validation Loss	0.0601	0.0569	0.0521	**0.0488**	0.0573
Batch Size	26	22	26	38	10
Learning Rate (l_rate)	0.000683	0.001075	0.000370	0.000691	0.000329
Recurrent Units (recurrent_units_0)	372	249	401	181	512
Dropout Rate 1 (dropout_rates_0)	0.167	0.219	0.113	0.159	0.100
Dropout Rate 2 (dropout_rates_1)	0.210	0.256	0.423	0.190	0.500
Convolution Filters (conv_filters_0)	122	91	81	109	21
Kernel Size (kernel_size_0)	6	3	5	4	3
Pool Size (pool_size)	5	5	3	4	4

Table 5.2: Summary of the best trials and optimal hyperparameters for different samplers achieved during final tuning

Figures 5.42 through 5.46 illustrate the slice plots for the best model training histories of all samplers.

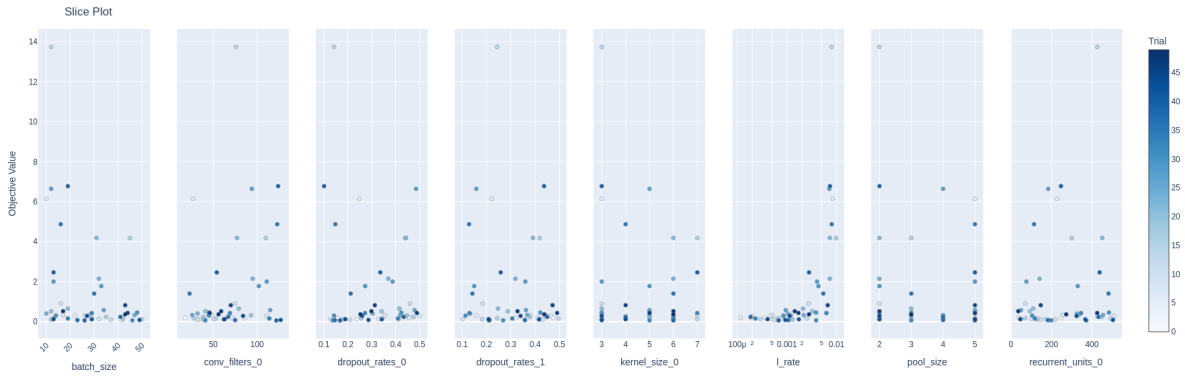

Figure 5.42: Final Tuning, Slice Plot - Random Sampler

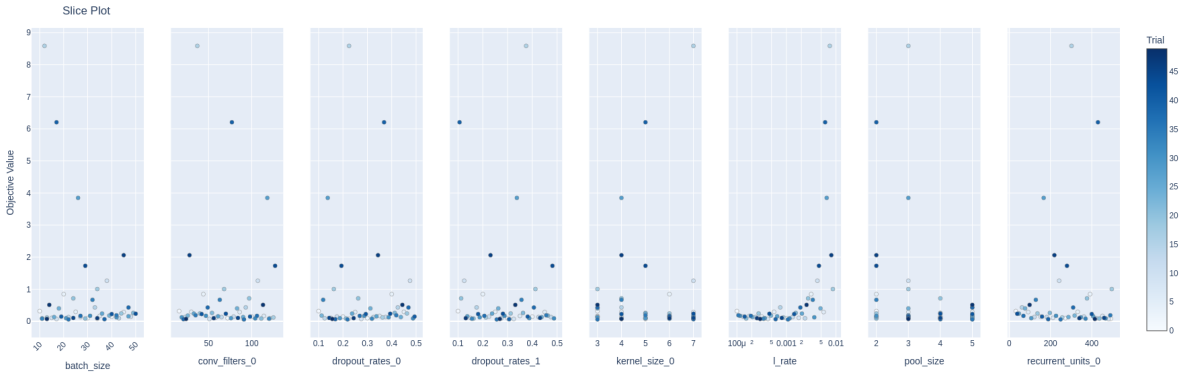

Figure 5.43: Final Tuning, Slice Plot - QMC Sampler

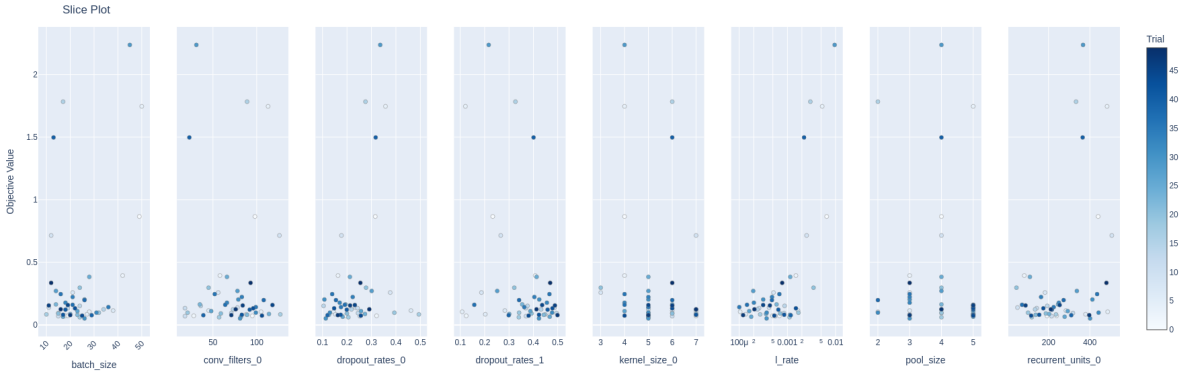

Figure 5.44: Final Tuning, Slice Plot - TPE Sampler

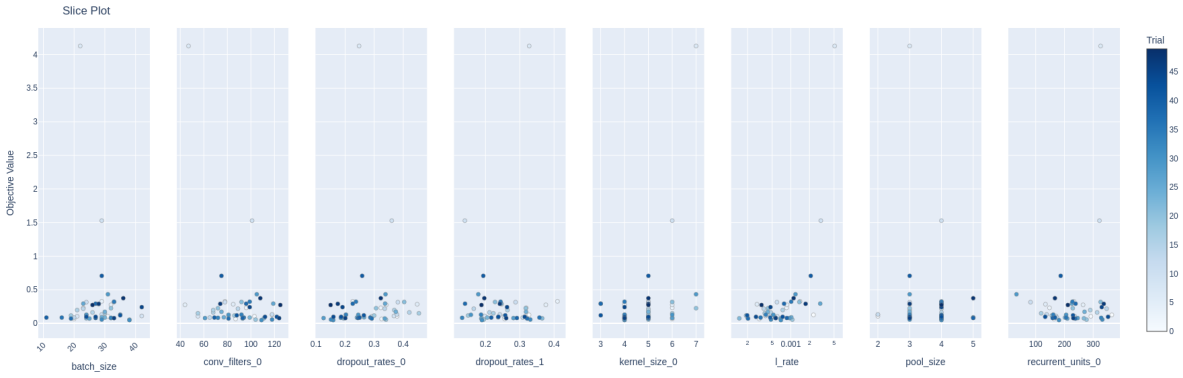

Figure 5.45: Final Tuning, Slice Plot - CMA-ES Sampler

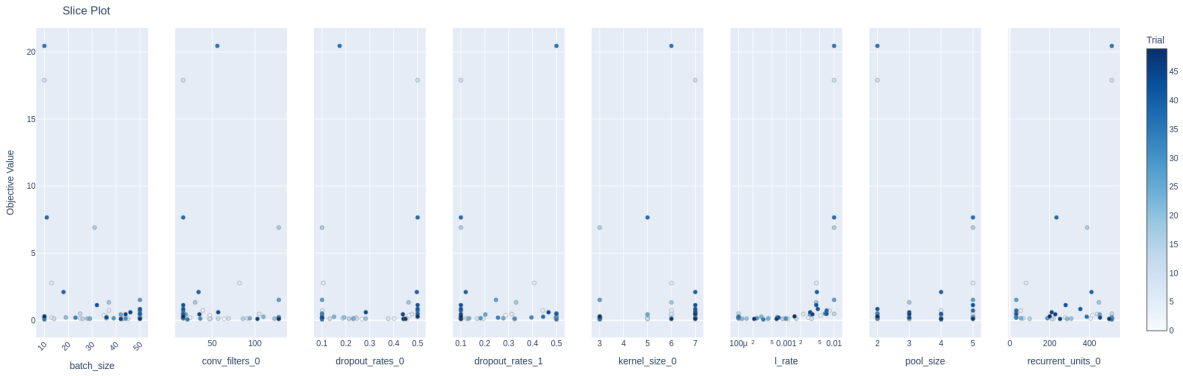

Figure 5.46: Final Tuning, Slice Plot - GP Sampler

5.5.2. Performance Analysis

The CMA-ES Sampler stands out as the most effective in tuning the model, evidenced by the tight clustering of lower objective values across multiple hyperparameters. TPE Sampler and QMC Sampler also showed decent performance but with less precision and slightly higher validation losses. The Random and GP Samplers exhibited the least effective optimization, with broader spreads and less targeted exploration of the hyperparameter space.

Figures 5.47 through 5.51 provides insight into how the models trained with the optimal hyperparameters. All models employed early stopping, the weights from their peak performances were restored.

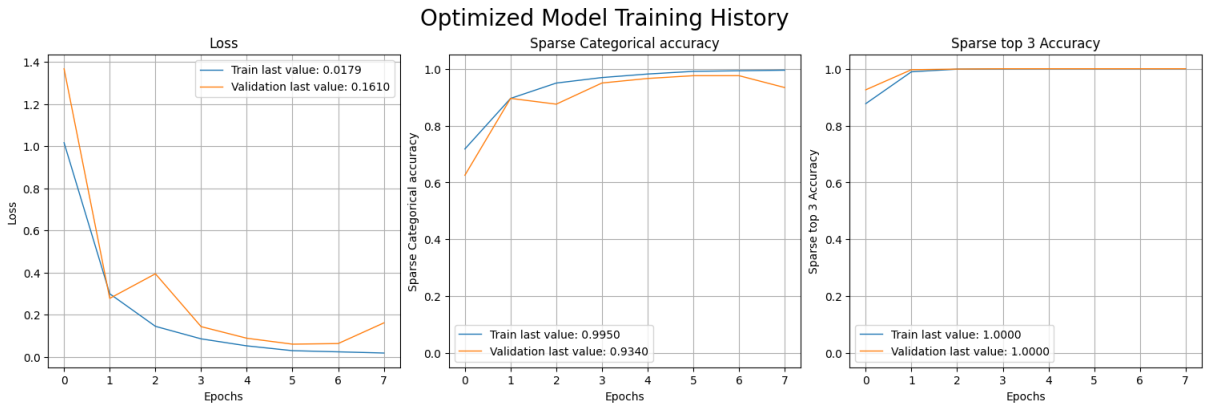

Figure 5.47: Final Model Training History - Random Sampler

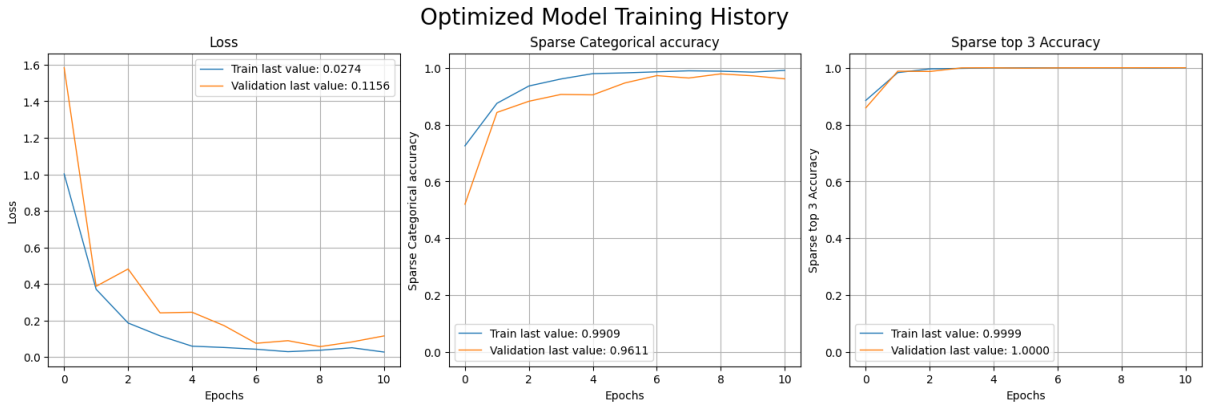

Figure 5.48: Final Model Training History - QMC Sampler

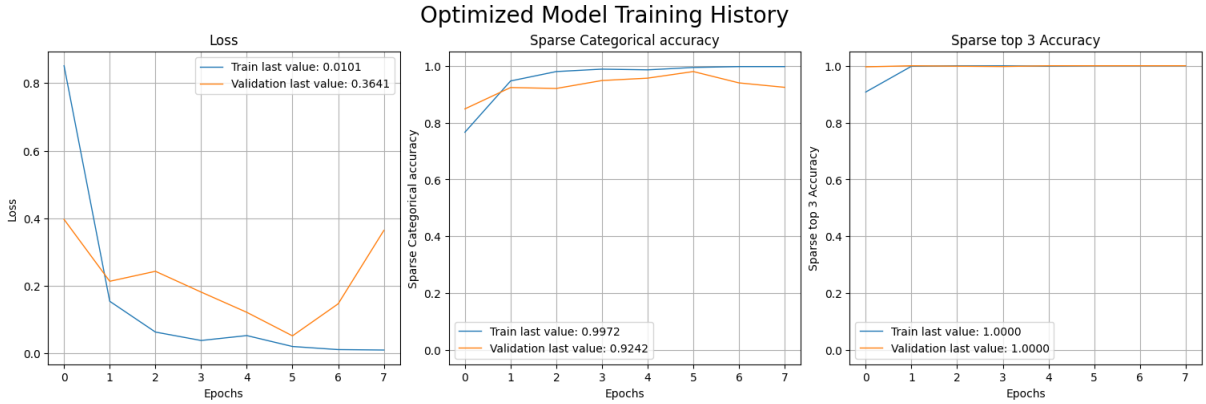

Figure 5.49: Final Model Training History - TPE Sampler

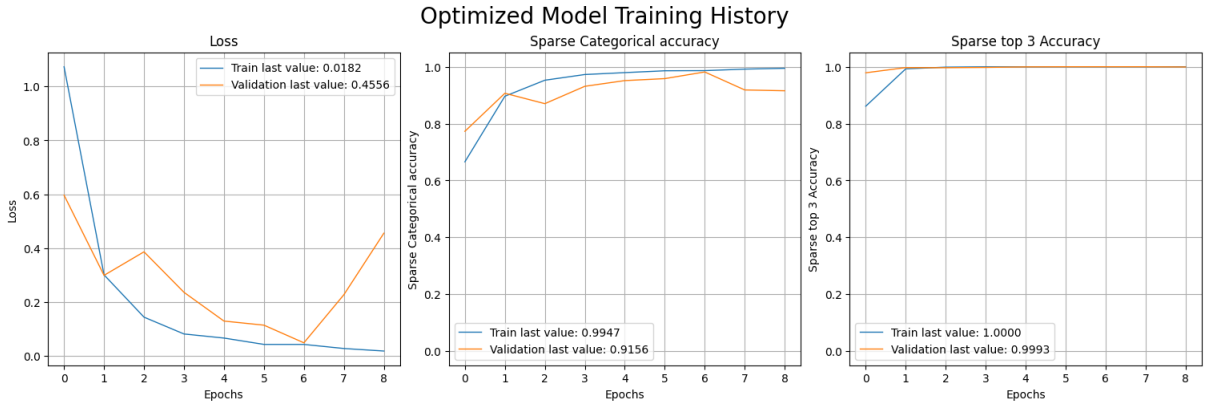

Figure 5.50: Final Model Training History - CMA-ES Sampler

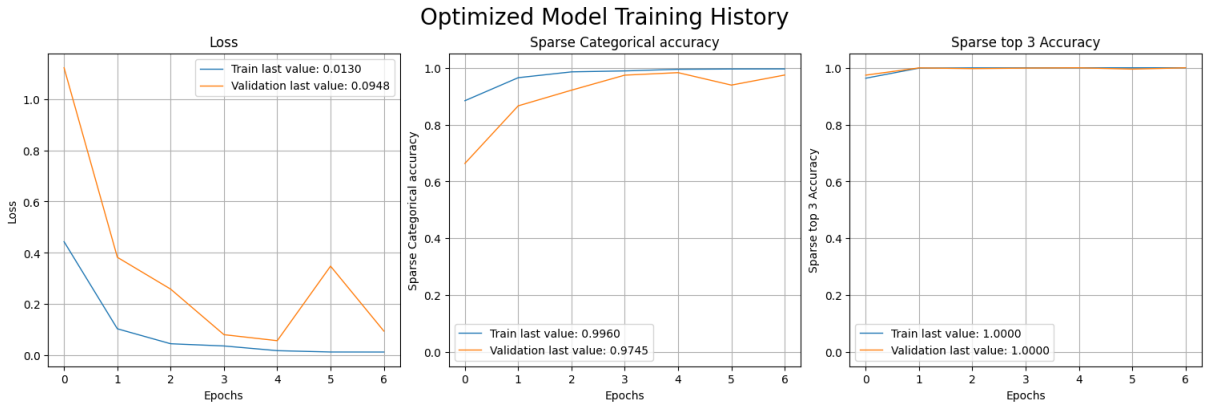

Figure 5.51: Final Model Training History - GP Sampler

The results of evaluation on the unseen data, the test set are presented in Figures 5.52 through 5.56.

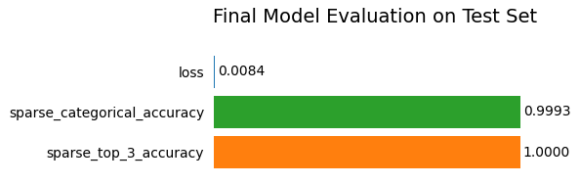

Figure 5.52: Best Model Evaluation - Random Sampler

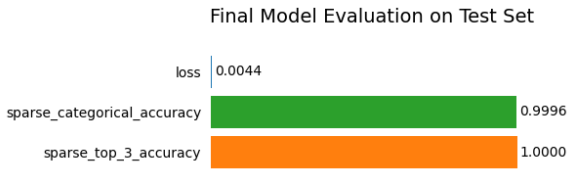

Figure 5.53: Best Model Evaluation - QMC Sampler

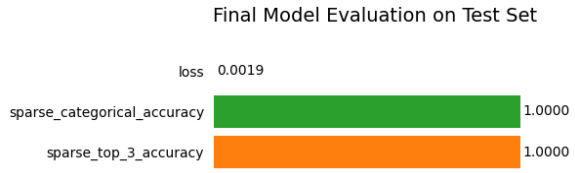

Figure 5.54: Best Model Evaluation - TPE Sampler

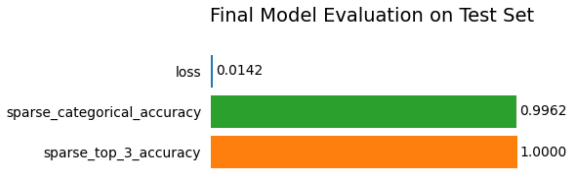

Figure 5.55: Best Model Evaluation - CMA-ES Sampler

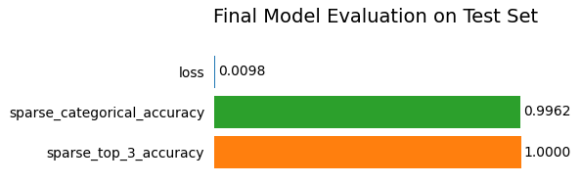

Figure 5.56: Best Model Evaluation - GP Sampler

Since all models achieved near-perfect accuracy on the test set, the CMA-ES Sampler model, which achieved the highest accuracy and lowest loss on the validation set at its peak performance, was chosen as the final model for this study.

6. Results and Discussion

The final chapter presents the key results of the research, which compares the baseline model with an optimized target model tuned by hyperparameters. This chapter examines the improvements in the accuracy of activity recognition, then discusses the practical implications of those findings for the HAR domain, and ends with a reflection on the limitations of this study and implications for future research.

6.1. Comparison with Baseline Model

The baseline model with simpler architecture was surpassed. The optimized model incorporated additional layers: Conv1D, ReLU, and MaxPooling. Such modifications enabled it to learn complex patterns presented within the studied dataset, hence boosting model's accuracy and lessening loss.

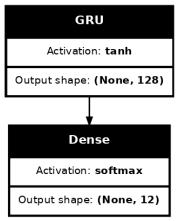

Figure 6.1: Baseline Model Architecture Comparison

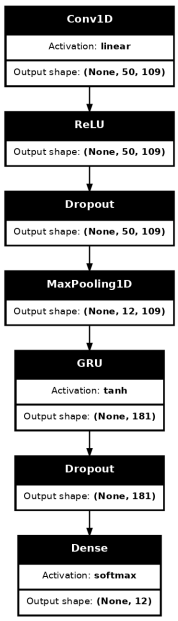

Figure 6.2: Final Target Model Architecture Comparison

The comparison of performance metrics shows the benefits of hyperparameter tuning. The comparison of these two evaluations on the test set was done using recordings from a subject whose data did not participate in either training or tuning.

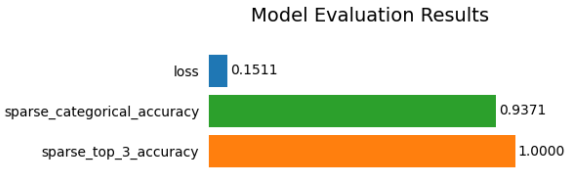

Figure 6.3: Baseline Model Evaluation

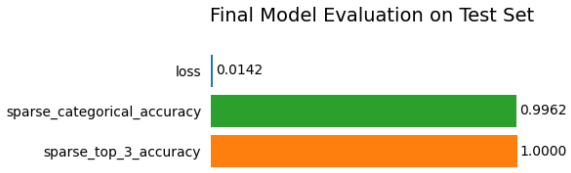

Figure 6.4: Final Model Evaluation

Figure 6.3 shows that the baseline model achieved an accuracy of 93.71% with a loss of 0.1511, while Figure 6.4 displays that the tuned model attained an accuracy of 99.62% with a loss of 0.0142. These experiments bring to light the far-reaching effect that the tuning process had on model performance.

6.2. Summary of Findings

The study demonstrates that targeted hyperparameter tuning and architectural enhancements leaded to significant improvements in the performance. The final model provides more reliable activity recognition, which is essential for practical applications in HAR. This research successfully developed a working model with strong accuracy, reinforcing the effectiveness of the approaches used. The findings confirm that combining advanced deep learning techniques with meticulous hyperparameter tuning can result in robust and dependable models suited to the demands of HAR applications.

6.3. Practical Applications

Although the reported enhancement in model accuracy is noteworthy, practical implications must be discussed: fitness tracking and rehabilitation. Improved model accuracy in recognizing activities within fitness-tracking applications forms one way of realizing better monitoring and feedback, thereby guiding users toward more successful fulfillment of their fitness objectives. Such a system would be invaluable during rehabilitation; it would classify physical activities accurately and ensure patients adhere to prescribed exercises that support their recovery process.

These applications certainly underline the importance of model improvement, but they are incremental rather than transformational. The successful development of a high-accuracy model for the respective applications speaks more to an evolutionary advance in ongoing HAR system development rather than a revolutionary breakthrough.

6.4. Limitations and Future Work

While the MHEALTH dataset is rich, it may not represent all possible scenarios that may take place in practice. In this regard, generalization performance for other datasets or different types of activities may end up being poor. The optimization process in the current study aimed at a set of hyperparameters and model architectures. Future studies may test a wider range of configurations and involve other deep learning architectures for general performance on diverse datasets.

Though the study did indeed improve model performance under a controlled environment, it raises further issues in terms of computational efficiency and responsiveness in real-time implementations in practice. Real-time setting deployment for such models calls for due consideration. Indeed, such models could see their improvements either by accounting for more diverse physiological metrics or additional data sources.

This study has demonstrated the effectiveness of hyperparameter tuning and model refinement in improving activity recognition models for HAR applications. Further research is needed in the quest of refining future HAR models, ensuring robustness, generality; testing in real-world applications.

Bibliography

[1] Shibo Zhang, Yaxuan Li, Shen Zhang, Farzad Shahabi, Stephen Xia, Yu Deng, and Nabil Alshurafa. Deep learning in human activity recognition with wearable sensors: A review on advances. *Sensors*, 22(4), 2022. https://www.mdpi.com/1424-8220/22/4/1476, Accessed: August 30, 2024.

[2] Brad E. Dicianno, Bambang Parmanto, Andrea D. Fairman, Theresa M. Crytzer, David X. Yu, Gede Pramana, David Coughenour, and Amanda A. Petrazzi. Perspectives on the evolution of mobile (mhealth) technologies and application to rehabilitation. *Physical Therapy*, 95(3):397–405, March 2015. Epub 2014 Jun 12. Retrieved on August 30, 2024, from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4757639/.

[3] 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, Deep Learning for Pattern Recognition, Accessed: August 30, 2024.

[4] Ling Bao and Stephen S. Intille. Activity recognition from user-annotated acceleration data. In *Pervasive Computing*. Springer, 2004. https://users.ece.cmu.edu/~bfrench/ewatch_files/papers/acceleromter%20activity%20recog/BaoIntille04.pdf, Accessed: August 30, 2024.

[5] Francisco J. Ordóñez and Daniel Roggen. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors*, 16(1), 2016. https://www.mdpi.com/1424-8220/16/1/115, Accessed: August 30, 2024.

[6] Oresti Banos, Rafael Garcia, and Alejandro Saez. Mhealth. UCI Machine Learning Repository, 2014. https://doi.org/10.24432/C5TW22, Accessed: August 30, 2024.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb), 2012. https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf, Accessed: August 30, 2024.

[8] Jun Qi, Po Yang, Atif Waraich, Zhikun Deng, Youbing Zhao, and Yun Yang. Examining sensor-based physical activity recognition and monitoring for healthcare us-

ing internet of things: A systematic review. *Journal of Biomedical Informatics*, 87:138–153, 2018. https://www.sciencedirect.com/science/article/pii/S153204641830176X, Accessed: August 30, 2024.

[9] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Sørensen Prentow, Mikkel Baun Kjærgaard, Anind K Dey, Thomas Sonne, and Mads Møller Jensen. Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2015. https://pure.au.dk/ws/files/93103132/sen099_stisenAT3.pdf, Accessed: August 30, 2024.

[10] Alessandro Ferrari, Davide Micucci, Maria Mobilio, and Paolo Napoletano. Deep learning and model personalization in sensor-based human activity recognition. *Journal of Reliable Intelligent Environments*, 9(1), 2023. https://link.springer.com/article/10.1007/s40860-021-00167-w, Accessed: August 30, 2024.

[11] Ming Zeng, Phuoc Nguyen, Bingjie Yu, Ole J. Mengshoel, Yingshu Zhu, Hengshu Wu, and Jian Zhang. Convolutional neural networks for human activity recognition using mobile sensors. In *2014 6th International Conference on Mobile Computing, Applications and Services*. IEEE, 2014. https://ieeexplore.ieee.org/document/7026300, Accessed: August 30, 2024.

[12] Charissa Ann Ronao and Sung-Bae Cho. Human activity recognition with smartphone sensors using deep learning neural networks. *Expert Systems with Applications*, 59:235–244, 2016. https://www.sciencedirect.com/science/article/pii/S0957417416302056, Accessed: August 30, 2024.

[13] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf, Accessed: August 30, 2024.

[14] Takuya Akiba, Shotaro Sano, Takeru Ohta, Toshihiko Yanase, and Manabu Koyama. Optuna: A next-generation hyperparameter optimization framework. *arXiv preprint arXiv:1907.10902*, 2019. https://arxiv.org/pdf/1907.10902, Accessed: August 30, 2024.

[15] Jia Wu, Xiu-Yun Chen, Hao Zhang, Li-Dong Xiong, Hang Lei, and Si-Hao Deng. Hyperparameter optimization for machine learning models based on bayesian optimization. *Journal of Electronic Science and Technology*, 17(1):26–40, 2019. https://www.sciencedirect.com/science/article/pii/S1674862X19300047, Accessed: August 30, 2024.

[16] Mhealth dataset. https://archive.ics.uci.edu/dataset/319/mhealth+dataset, Accessed: August 30, 2024.

[17] Oresti Banos and Rafael García. mhealthdroid: A novel framework for agile development of mobile health applications. In *International Workshop on Ambient Assisted Living*, volume 8868, pages 91–98, 2014. https://www.researchgate.net/publication/273763583_mHealthDroid_A_Novel_Framework_for_Agile_Development_of_Mobile_Health_Applications, Accessed: August 30, 2024.

[18] PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/. Accessed: 2023-08-30.

[19] SQLAlchemy Authors. Sqlalchemy: The database toolkit for python. https://www.sqlalchemy.org/. Accessed: 2023-08-30.

[20] TensorFlow Developers. Tensorflow: An open source machine learning framework for everyone. https://www.tensorflow.org/. Accessed: 2023-08-30.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997. https://www.bioinf.jku.at/publications/older/2604.pdf, Accessed: August 30, 2024.

[22] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014. https://arxiv.org/abs/1412.3555, Accessed: August 30, 2024.

[23] Jason Brownlee. Convolutional layers for deep learning neural networks, 2020. https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/, Accessed: August 30, 2024.

[24] Hassan Ismail Fawaz and Lhassane Idoumghar Pierre-Alain Muller Germain Forestier, Jonathan Weber. Deep learning for time series classification: A review. *Data Mining and Knowledge Discovery*, 33(4):917–963, mar 2019. https://arxiv.org/pdf/1809.04356, Accessed: August 30, 2024.

[25] Bendong Zhao, Jiaqi Lu, Shanghang Chen, Jun Liu, and Dongyao Wu. Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics*, 28(1), 2017. https://ieeexplore.ieee.org/document/7870510?denied=, Accessed: August 30, 2024.

[26] Gil Keren and Björn Schuller. Convolutional rnn: an enhanced model for extracting features from sequential data. 2016. https://arxiv.org/pdf/1602.05875, accessed 30 August 2023.

[27] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks, 2015. https://arxiv.org/abs/1507.06228, Accessed: August 30, 2024.

[28] Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline, 2016. https://arxiv.org/abs/1611.06455, Accessed: August 30, 2024.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org, Accessed: August 30, 2024.

[30] Nils Y. Hammerla, Shane Halloran, and Thomas Ploetz. Deep, convolutional, and recurrent models for human activity recognition using wearables, 2016. https://arxiv.org/abs/1604.08880, Accessed: August 30, 2024.

[31] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 351–360, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. https://doi.org/10.1145/3038912.3052577, Accessed: August 30, 2024.

[32] Rmsprop optimization method. https://paperswithcode.com/method/rmsprop, Accessed: August 30, 2024.

[33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. https://arxiv.org/abs/1412.6980, Accessed: August 30, 2024.

[34] Timothy Dozat. Incorporating nesterov momentum into adam. *ICLR Workshop*, 2016. https://openreview.net/pdf/OM0jvwB8jIp57ZJjtNEZ.pdf, Accessed: August 30, 2024.

[35] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 1994. https://www.comp.hkbu.edu.hk/~markus/teaching/comp7650/tnn-94-gradient.pdf, Accessed: August 30, 2024.

[36] Josef Dick and Friedrich Pillichshammer. *Digital Nets and Sequences: Discrepancy Theory and Quasi-Monte Carlo Integration*. Cambridge University Press, 2010. https://web.maths.unsw.edu.au/~josefdick/preprints/DP_book_preprint.pdf, Accessed: August 30, 2024.

[37] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. In *Evolutionary Computation*, volume 9. MIT Press, 2001. http://www.cmap.polytechnique.fr/~nikolaus.hansen/cmaartic.pdf, Accessed: August 30, 2024.

[38] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012. https://arxiv.org/abs/1206.2944, Accessed: August 30, 2024.

List of Equations

List of Tables

List of Figures