POLITECNICO DI TORINO

MASTER's Degree in Mechatronic Engineering

MASTER's Degree Thesis

Recurrent Neural Networks for Driver Drowsiness Detection

Supervisors

Prof. Massimo VIOLANTE

Prof. Luigi PUGLIESE

Candidate

Catia Sofia GIANNUZZI

2023-2024

Summary

The focus of this thesis is to probe the risk of reverse engineering applied to the PredictS algorithm, property of Sleep Advice Technologies Srl, which takes as inputs some physiological data sampled at 1Hz frequency from the Garmin smartwatches and computes the drowsiness level of the driver, while driving, in order to predict sleep events, with some minutes of advantage. This approach is especially effective in cases of long trip distances and straight roads.

Since we're dealing with time-dependent sequences of data I opted for taking into consideration the recurrent neural network, RNNs, which is one of the most suitable for sequence data analysis (for example texts or time series). In fact, they have connections that form directed cycles, allowing them to exhibit dynamic temporal behavior.

I put my attention on its variants GRU and LSTM, with more focus on the first one, which is less computationally complex than the second one, even if maintaining similar characteristics.
GRU and LSTM models are commonly used to address the vanishing gradient problem, which is a relevant issue especially in deep neural networks, that have gradients which diminish rapidly (sigmoid an hyperbolic tangent functions). A consequence of this phenomenon is that deep neural networks fail to learn meaningful representations from the data, especially in early layers, and this limits the ability of the network to generalize well unseen data.

Once probing the maximum performances that can be obtained by training a RNN, to replicate the behaviour of the PredictS software (I've reported the plots to notice graphically if the drowsiness levels are predicted or not by the model), the next step is trying to insert a random jitter in the detection of the sleep events, to exploit if this can affect the training, reducing the metrics values, without deteriorating the reliability of the algorithm, since it will be sold as product, and has to be compliant to certain medical specifications.

At the end of this second part, I've implemented the best model found into an Android application. The used technologies include the Android Studio tool and the programming languages that can be used to carry it out are Kotlin and Java.
Once deployed the application on the smartphone, I've tested its performances compared with the ones of the PredictS software, in order to investigate the feasibility of this approach in an hardware environment, in terms of memory occupation and battery consumption.

Acknowledgements

I would like to express my deepest gratitude to all those who have supported me throughout the journey of completing this thesis.

First and foremost, I am profoundly grateful to my supervisor, Prof.Massimo Violante, for their invaluable guidance, patience, and insightful feedback, which shaped this work. Their encouragement and dedication have been a constant source of motivation.

I am also thankful to my co-supervisor, Prof.Luigi Pugliese, for their time, constructive comments, and thoughtful suggestions that helped improve this research.

Special thanks to my colleagues and friends, who provided not only technical advice but also moral support throughout the challenging moments. The stimulating discussions and shared experiences made this journey much more manageable and enjoyable.

I owe a debt of gratitude to my family for their unwavering support and belief in me. Their love, encouragement, and patience have been the foundation of my achievements.

Finally, I wish to acknowledge the financial and institutional support provided by Politecnico di Torino and Sleep Advice Technologies Srl that made this research possible.

This thesis would not have been possible without the contribution and support of all these wonderful individuals.

Table of Contents

Acronyms VI

1 Introduction 1
1.1 Driver Assistance Systems and Autonomous vehicles . . . 1
1.2 Sleep during driving . . . 2

2 State of the art 5
2.1 Sleep Prediction Algorithms . . . 5
2.1.1 Vehicular-based Techniques . . . 5
2.1.2 Behavioral-based Techniques . . . 6
2.1.3 Physiological-based Techniques . . . 7
2.1.4 Hybrid Techniques . . . 8
2.2 Physiological-based Techniques . . . 9

3 AI for sleep prediction 11
3.1 First reasoning about the type of problem . . . 11
3.2 Feed-Forward Neural Networks and Backpropagation . . . 12
3.3 Recurrent Neural Networks . . . 14

4 Data analysis and pre-processing 15
4.1 Dataset: experimental activities . . . 15
4.1.1 Data analysis and filtering . . . 17
4.1.2 Data processing . . . 21
4.1.3 Z-score normalization of data . . . 25
4.1.4 Scheduler function . . . 26
4.1.5 Cross-validation method . . . 27

5 GRU model 31
5.1 Method B and C: training with cross-validation . . . 33
5.2 Method B and C: training achievements . . . 36
5.2.1 Window size of 300 and row input data . . . 36
5.2.2 Window size of 300 and row input data - filtering of temporal windows 38
5.2.3 Window size of 300 and pre-processed input data . . . 40
5.2.4 Window size of 300 and pre-processed input data - validation with 2 datasets . . . 49
5.2.5 Window size of 240 and pre-processed input data - validation with 2 datasets . . . 49
5.2.6 Window size of 128 and pre-processed input data - validation with 2 datasets . . . 50

5.3 Age as an additional feature 51
5.4 Method A: handle unbalanced data 55

6 LSTM model **61**
6.1 Method 2: window size of 300 and row input data 62
6.2 Techniques to handle unbalancing of data 64

7 Introduction of a random jitter **67**
7.1 Delay in predicting FATIGUED and DROWSY 68
7.1.1 Method B 68
7.1.2 Method A 70
7.2 Delay in predicting higher levels of drowsiness and in restoring the lower ones 71
7.3 Delay in predicting levels of drowsiness and revise the data related to the FATIGUED level 73

8 Implementation of the model **77**
8.1 Set-up 77
8.1.1 TensorFlow Lite model 77
8.1.2 Behaviour of the class Prediction: management of input data and predictions 80
8.2 Testing of the application in terms of prediction 84
8.3 Memory usage and battery consumption 87
8.3.1 Memory usage 87
8.3.2 Battery consumption 91

9 Conclusions **93**

Acronyms

DAS	Driver Assistance Systems
ABS	Antilock Braking Systems
TCS	Traction Control Systems
ESC	Electronic Stability Control
ADAS	Advanced Driver Assistance Systems
SAE	Society of Automotive Engineers
HR	Heart Rate
HRV	Heart Rate Variability
PSG	Polysomnography
EEG	Electroencephalogram
ECU	Electronic control units
SWA	Steering Wheel Angle
SVM	Support Vector Machine
KNN	K-Nearest Neighbor
ORD	Observer Rating of Drowsiness
EAR	Eye Aspect Ratio
KSS	Karolinska Sleepiness Scale
PVT	psychomotor vigilance task
DDD	Driver Drowsiness Detection
PERCLOS	Percentage of Eyelid Closure
RF	Random Forest

EDA	Electrodermal Activity
ECG	Electrocardiography
RRI	RR intervals
ANS	Autonomic Nervous System
EOG	Electrooculography
PPG	Photoplethysmography
SNS	Sympathetic Nervous System
PSNS	Parasympathetic Nervous System
SpO2	Saturation of peripheral oxygen
SDK	Software Development Kit
rKSS	Reduced Karolinska Sleepiness Scale
sw	Software
NN	Neural Network
AI	Artificial Intelligence
TP	True positive
TN	True negative
FP	False positive
FN	False negative
FFNN	Feed-Forward Neural Network
RNN	Recurrent Neural Network
GRU	Gated Recurrent Units
LSTM	Long Short-Term Memory

Chapter 1

Introduction

1.1 Driver Assistance Systems and Autonomous vehicles

The integration of technology into the automotive industry has been pivotal in evolving Driver Assistance Systems (DAS), enhancing vehicle safety and transforming the driving experience.
Since the 1950s with the introduction of Early Warning Systems, which provided foundational warnings related to driving conditions and speed, DAS were improved placing future advancements, with the introduction of cruise control in the 1960s, allowing drivers to set a constant speed. Antilock Braking Systems (ABS) was deployed in the 1980s and become an essential system in avoiding wheel lock-up during braking, drastically improving vehichle safety and control.
Further advancements like Traction Control Systems (TCS) and Electronic Stability Control (ESC) were introduced in the 1990s, technologies that from one side prevent loss of traction, crossing the engine power, torque with the actual road surface characteristics, from the other detect loss of traction from the loss of steering control and activate the brake automatically to induce steer the wheels where the driver is intended to.
The 21st century brought Advanced Driver Assistance Systems (ADAS) with features like adaptive cruise control, collision avoidance and lane departure warning, which laid the foundation for semi-autonomous driving. [1, 2]

Autonomous vehicle development has followed the Society of Automotive Engineers (SAE) levels of automation, starting with basic driver assistance features and advancing towards higher levels of automation.
Operating primarily at SAE Levels 1 and 2, these systems introduced key features such as adaptive cruise control, automated parking and lane-keeping assistance. This period marked the beginning of semi-autonomous driving capabilities.

The 2010s saw substantial advancements in automation, particularly with the development of SAE Level 3 autonomous vehicles. Level 3 systems, characterized by conditional automation, allow vehicles to handle most driving tasks in certain conditions. Despite this progress, challenges related to regulatory frameworks, safety concerns, and the complexities of human-machine interaction have slowed the widespread adoption of these systems. In recent years, there has been a strong push towards achieving higher levels of automation, specifically SAE Levels 4 and 5. These levels aim to enable vehicles to operate

autonomously under a wider range of conditions and environments. However, fully automated systems have not yet been realized, and there remains a need for drivers to take control in specific scenarios.
Currently, most vehicles operate at SAE Levels 2 or 3, where the driver is still required to ensure safety by remaining attentive and ready to take control of the vehicle when necessary. [3]

In the pursuit of achieving higher levels of vehicle autonomy, the integration of Human-Centric Advanced Driver Assistance Systems (ADAS) becomes essential. These systems employ physiological monitoring, driver behavior analysis, and adaptive feedback mechanisms to help drivers maintain situational awareness, especially for predicting and preventing drowsiness at the wheel and readiness to take control when necessary.
These systems aim to improve the interaction between the driver and the vehicle, ensuring that the driver remains engaged and informed, even as the vehicle assumes more driving responsibilities.

The emphasis on human-centric systems highlights the importance of understanding driver behavior, cognitive load, and emotional state. By integrating advanced sensors and machine learning models, these systems can anticipate and mitigate potential risks, offering a safer and more intuitive driving experience.

1.2 Sleep during driving

In every country, road traffic accidents are a major public health problem and cause huge societal and financial burdens.
About 1.3 million deaths occur each year as a result of road traffic accidents globally, causing a 3% loss of the gross domestic product of most countries.
The US National Highway Traffic Safety Administration has estimated that worldwide every year, about 100,000 road accidents are caused by drowsiness, accounting for > 1500 deaths and $> 70{,}000$ injuries [4].

Several studies during the last 20 years have suggested that sleepiness is among the main factors that cause road traffic accidents. Sleepiness results in disrupted brain functioning, such as reduced reaction time or decreased ability for decision-making and it is a major contributor to road traffic accidents, which often occur when a driver experiences drowsiness at the wheel, or due to sleep abnormalities, lack of sleep, alcohol consumption or medication. Furthermore, it causes disruption of neurological functions. [4].

Sleepiness while driving is a significant factor in road traffic accidents worldwide, contributing to an estimated 3% to over 30% of such incidents [5] [6]. These accidents can result from various sleep-related conditions or simply from insufficient sleep [7] [8]. Over 20% of drivers report needing to stop driving at least once due to feelings of drowsiness.

Sleep prediction plays a crucial role in Human-Centric Advanced Driver Assistance Systems (ADAS), particularly when it comes to detecting driver drowsiness and implement preventive measures. Drowsiness severely impacts driving performance and significantly raises the likelihood of accidents. Advanced algorithms designed to predict sleep use physiological data, such as heart rate variability (HRV), electroencephalograms (EEG), and photoplethysmography (PPG), to identify early signs of fatigue and alert the driver promptly.

By continuously tracking these physiological signals, the system can foresee the onset of sleep and take preventive action, such as issuing alerts or modifying vehicle settings to enhance driver attentiveness. Incorporating sleep prediction into Human-Centric ADAS improves road safety and aids in the shift toward higher levels of vehicle autonomy by ensuring the driver stays active and aware.[9]

In both academic research and the evolving market, various strategies have emerged to address the widespread issue of drowsy driving. These strategies generally fall into two main categories:

1. **Prevention:** A proactive approach focuses on continuously monitoring a person's health status during sleep. Recognizing the multiple factors contributing to poor sleep quality, such as diet and stress, highlights the importance of preventive measures. By understanding these lifestyle factors, targeted actions can be taken to improve sleep quality, reducing the risk of drowsiness while driving.

2. **Prediction:** The predictive approach involves real-time observation of a driver's physiological state and behavior to detect signs of impending drowsiness. This forward-looking strategy enables timely interventions, ensuring a safer driving experience. Indicators such as physiological data and steering behavior are crucial in forecasting and addressing drowsiness before it becomes a serious hazard.

The purpose of this thesis is to study how to afford the problem of tracking sleep events through the use of a data-driven approach: the purpose is predicting sleep events with some minutes of advantage, all by only analyzing physiological variables collected from wearable smartwatch.

Chapter 2

State of the art

2.1 Sleep Prediction Algorithms

To conduct a comprehensive analysis of the state of the art in driver monitoring systems aimed at predicting sleepiness, this chapter will examine four macro-fields: Vehicular-based Techniques, Behavioral-based Techniques, Physiological-based Techniques, and Hybrid Techniques.
Each of these categories offers a distinct approach to monitoring and predicting driver drowsiness, leveraging different types of data and methodologies.

2.1.1 Vehicular-based Techniques

Vehicle-based techniques represent a groundbreaking and extensively studied approach to detecting driver drowsiness. By utilizing data generated from modern vehicles, these methods aim to identify patterns and irregularities that suggest a driver may be fatigued [10]. Techniques here focus on analyzing driving patterns, such as:

- Steering wheel movements: unusual steering patterns, such as frequent corrections or abrupt changes, can indicate a loss of alertness.
- Lane keeping: deviations from the lane without corrective action are a strong indicator of drowsiness.
- Speed variations: inconsistent speed, either slowing down unexpectedly or erratic acceleration, may signal fatigue.
- Braking patterns: sudden or delayed braking could suggest a decline in the driver's reaction time due to sleepiness.

These methods often rely on data collected from in-vehicle sensors and the car's electronic control units (ECUs). Advanced systems may also incorporate machine learning algorithms to detect patterns that correlate with drowsiness. The primary advantage of vehicular-based techniques is their non-intrusiveness, as they do not require any direct interaction with the driver. However, their effectiveness can vary depending on external conditions such as road type and traffic.

In this field studies as Ma et al. [11] developed a system that detects drowsiness using lateral distance, which is derived from lane curvature, position, and curvature derivatives.

Data were collected using a portable instrumentation system with a video camera mounted on the vehicle's front bumper. This setup captured real-time footage of both the roadway and the driver's face, creating a comprehensive dataset for vehicle movement. The data were analyzed using wavelet transforms, Support Vector Machines (SVM), and Neural Networks (NN), achieving an accuracy rate of over 90%.
Li et al. [12] introduced a different approach, utilizing SWA data to identify driver fatigue. They gathered SWA data over nearly 15 hours of real driving, using a sliding window to extract approximate entropy from the SWA time series. These entropy features were then linearized with adaptive piecewise linear fitting to measure deviations and calculate the warping distance between series, which helped assess the driver's alertness. A custom binary decision classifier was used to distinguish between 'drowsy' and 'alert' states, with an accuracy of 84.85% in detecting drowsiness and 78.01% for identifying alertness.

These studies highlight the effectiveness of vehicle-based measures, as they directly reflect the driver's cognitive state through vehicle control and movement data.

2.1.2 Behavioral-based Techniques

[13] [14] While vehicle-based techniques provide valuable insights into a driver's condition by leveraging data collected directly from the vehicle, behavioral-based techniques complement this by focusing on the driver's actions and movements during driving. Key indicators in this category include:

- Eye Movement Tracking: the frequency and duration of blinks, as well as gaze direction, are critical factors. Extended eyelid closures (microsleeps) are a direct sign of impending sleep.
- Head Position Monitoring: nodding or head tilting is often associated with the early stages of sleep onset.
- Facial Expressions: yawning, changes in facial muscle tension, and other expressions can be captured and analyzed to assess alertness.
- Response Time: delayed reactions to external stimuli, such as traffic signals or sudden obstacles, can indicate reduced alertness.

Behavioral-based techniques bridge the gap between vehicular-based and physiological-based methods by analyzing body movements captured in video footage. As noted by Albadawi et al. [15], these techniques can be divided into two categories: Thermal Imaging-Based Systems and Image-Based Systems.

For example, Kiashari et al. [16] developed a thermal imaging system capable of detecting and analyzing a driver's respiration patterns. By calculating metrics like the average and standard deviation of respiration rates and the ratio of inhalation to exhalation time, they were able to identify drowsiness using two machine learning models: Support Vector Machine (SVM) and K-Nearest Neighbor (KNN). The SVM model outperformed the others, achieving 90% accuracy, 85% specificity, 92% sensitivity, and 91% precision when compared to the Observer Rating of Drowsiness (ORD), which is a non-intrusive assessment based on judgments from multiple human observers [17].

Image-Based Systems are another subcategory, focusing on eye, head, and mouth movements. For instance, Maior et al. [18] implemented real-time eye movement analysis by tracking blinks using the Eye Aspect Ratio (EAR), which measures the height-to-width ratio of the eye. Consecutive EAR values were used as input for machine learning algorithms, generating outputs such as the Psychomotor Vigilance Test (PVT) and Karolinska Sleepiness Scale (KSS). These metrics help assess levels of subjective drowsiness and performance disruptions like temperature or pressure changes. Among the tested classification methods, SVM again proved most effective, achieving a 94.9% accuracy rate in relation to KSS and PVT metrics.

In another study, Bamidele et al. [19] introduced a non-intrusive Driver Drowsiness Detection (DDD) system that monitored facial and eye movements using video data from the NTHU DDD Computer Vision Lab. Their approach involved data acquisition and preprocessing, followed by feature extraction, including PERCLOS (Percentage of Eyelid Closure) and blink frequency. Various classifiers, such as KNN and SVM, were used to determine whether the extracted features indicated drowsiness or alertness. The models were validated using objective indicators like head nodding and slow blinking.

Zandi et al. [20] proposed another non-intrusive drowsiness detection system based on eye-tracking data collected from simulated driving environments. They extracted 34 features from the eye-tracking signals and tested two binary classifiers: Random Forest (RF) and non-linear SVM. The RF classifiers demonstrated accuracy rates between 88.37% and 91.18% across all trials, while the SVM classifiers ranged from 77.12% to 82.62%, highlighting the effectiveness of eye-tracking data for detecting drowsiness.

Finally, Celecia et al. [21] developed an affordable, portable DDD device that integrates features from both eye and mouth analysis. Their system, built on a Raspberry Pi 3 Model B, utilized infrared imaging and metrics like PERCLOS and eye-closing duration to detect drowsiness. The combination of these features resulted in a robust system with a 95.5% accuracy rate across varying lighting conditions.

As explained previously these methods often employ cameras and computer vision algorithms to continuously monitor the driver's behavior. While highly effective, these techniques can be intrusive and may require more sophisticated hardware, potentially leading to privacy concerns.

2.1.3 Physiological-based Techniques

These techniques focuses on real-time monitoring of the driver's physiological signals to assess their state of alertness. Commonly monitored physiological parameters include:

- Electrocardiography (ECG) tracks the heart's electrical activity.
- Photoplethysmography (PPG) measures changes in blood volume in the microvascular tissue.
- Heart Rate (HR) and Heart Rate Variability (HRV): fluctuations in heart rate and its variability can reflect the level of fatigue or stress [22].
- Electrodermal Activity (EDA): changes in skin conductance, which are influenced by sweat gland activity, can indicate stress levels and alertness [22].
- Electroencephalography (EEG): direct monitoring of brain activity can provide the most accurate measure of sleep onset, though it is more invasive [23].
- Breath Rate and Body Temperature: these parameters can fluctuate with the driver's level of fatigue and provide additional data points for drowsiness detection.

Physiological monitoring provides direct and accurate measurements of the driver's state, but it often requires wearable devices or sensors that might be uncomfortable or invasive for long-term use.

Heart Rate Variability (HRV), which refers to the fluctuation in time between consecutive heartbeats, has been studied by Lee et al. to evaluate the effectiveness of wearable ECG and PPG sensors in detecting driver drowsiness by monitoring HRV signals. The study utilized different types of recurrence plots (RPs) to analyze R-R intervals (RRI) from heartbeats, including binary recurrence plot (Bin-RP), continuous recurrence plot (Cont-RP), and thresholded recurrence plot (ReLU-RP), which applies a modified rectified linear unit function to filter Cont-RP. ReLU-RP was found to provide the most distinct and reliable patterns for differentiating between awake and drowsy states. PPG signals had an accuracy of 64%, with precision at 71%, recall at 78%, and an F-score of 71%. ECG signals performed slightly better, achieving 70% accuracy, 71% precision, 85% recall, and an F-score of 77% [24].

Koh et al. (2017) used PPG signals to detect drowsiness by analyzing high frequency (HF), low frequency (LF), and the LF/HF ratio. Their method, tested using sensors placed on participants' fingers and earlobes during a driving simulation, found that LF, HF, and LF/HF values significantly differed between awake and drowsy states [25].

Concerning drowsiness Detection Using Wrist-Worn Sensors, Kundinger et al. (2020) proposed a system utilizing physiological data from wrist-worn sensors to detect drowsiness by analyzing HRV and correlating it with autonomic nervous system (ANS) activity. Among the machine learning models tested, the K-Nearest Neighbors (KNN) algorithm achieved the highest accuracy at 92.13% [26].

Fujiwara et al. (2018) developed an HRV anomaly analysis algorithm to detect drowsiness, based on the premise that alertness affects the autonomic nervous system and HRV. By analyzing changes in HRV features like the mean and standard deviation of RRI, total power, and adjacent RRI pairs spaced by more than 50 ms, the algorithm achieved 92% accuracy [27].

Dealing with the topic of drowsiness Detection via Respiratory Signals, Guede-Fernandez et al. (2019) introduced an algorithm that detects drowsiness by analyzing respiratory rate variability (RRV) through respiratory inductive plethysmography band sensors. Their system, which tracks variations in respiratory signals, produced a drowsiness index with 90.3% sensitivity and 96.6% specificity [28].

While behavioral- and vehicle-based methods show potential, they face challenges in real-world settings. As noted by Razman et al., physiological-based techniques demonstrate superior accuracy in diverse conditions, such as varying lighting, autonomous driving scenarios, and when subjects are wearing sunglasses.

2.1.4 Hybrid Techniques

Their aim is to combine the strengths of vehicular, behavioral, and physiological approaches to provide a more comprehensive and accurate assessment of driver sleepiness [29] [30]. By integrating multiple data sources, these techniques can:

- Enhance Robustness: reduce the reliance on any single type of data, making the system more adaptable to different driving conditions and individual differences among drivers.
- Improve Accuracy: use machine learning models that analyze cross-correlations between different data types to improve prediction accuracy.

- Offer Flexibility: provide a tailored approach depending on the available data and specific requirements of the driving scenario.

Hybrid systems can be more complex and costly to implement due to the need for multiple sensors and advanced data fusion techniques. However, they offer the most potential for reliable and accurate driver drowsiness detection.

2.2 Physiological-based Techniques

In the field of sleep medicine, electroencephalography (EEG) recordings are essential for sleep scoring, as sleep onsets and stages are determined through EEG data. Despite the development of EEG-based drowsiness detection methods [31]–[32], accurately recording EEG during driving poses challenges due to its sensitivity to motion artifacts and the significant physical restrictions it imposes. Consequently, various alternative driver drowsiness detection systems that do not rely on EEG have been created.

These systems often analyze driver facial images and vehicle travel data to detect drowsiness [33], and are under the denomination of vehicular-based techniques. However, these approaches require the installation of specialized equipment in vehicles, such as cameras for capturing facial images or data logging devices to access travel data.

Physiological-based techniques can be considered another macro-field in driver monitoring system for predict sleepiness, which utilize real-time data from bodily functions, such as heart rate and blood volume (measured by PPG and ECG), to understand a person's state. This data provides valuable insights into the autonomic nervous system (ANS), which unconsciously regulates essential bodily processes.

The ANS comprises two main parts:

1. **Parasympathetic Nervous System (PSNS)**: associated with relaxation and recovery
2. **Sympathetic Nervous system (PNS)**: linked to stress and activity

As sleep approaches, the body transitions from a dominant sympathetic state to a parasympathetic one. This change in bodily functions is essential for accurately determining sleep onset and quality. Consequently, physiological signals are crucial for monitoring sleep patterns and detecting drowsiness.

Heart rate variability (HRV), defined as the fluctuation in RR intervals (RRI) observed in an electrocardiogram (ECG), is a widely recognized physiological phenomenon that indicates the activity of the autonomic nervous system (ANS) [34]. When HRV is high, the parasympathetic nervous system is dominant, while low HRV indicates dominance of the sympathetic nervous system. In addition several studies have reported changes in HRV associated with sleep stage transitions [35]: HRV measurements tend to increase while sleeping and decreased upon waking [36].

Photoplethysmography (PPG) is a non-invasive method that measures changes in blood volume within tissues. It works by shining light on the skin and detecting the amount of light absorbed or reflected. When blood flows through the tissue, it absorbs more light, and when it flows away, less light is absorbed.

By tracking these changes in light, PPG can be used to monitor vital signs like heart rate and blood oxygen levels. The resulting waveform shows rhythmic patterns that correspond to the heart's pumping action and breathing [37].

Commercial smartwatches devices transform raw PPG to obtain physiological parameters, such as HR, HRV and SpO2, which can be managed by the user through smartphone apps, where the data are sent wirelessly.

For the purposes of PredictS software, the algorithm developed by SAT Srl to monitor driver drowsiness, Garmin wearable devices are employed to collect physiological data with a sampling frequency of 1Hz, accessible via the Garmin Health SDK [38].
The data are analyzed in sliding windows of size *num_samples* samples to identify patterns in heart rate and heart rate variability changes. These patterns are used to classify a person's alertness level into five categories (calibration, awake, low and high drowsiness) based on a simplified sleepiness scale [39].

Chapter 3

AI for sleep prediction

3.1 First reasoning about the type of problem

The problem we're dealing with in this research involves the use of Supervised learning, in the view of making the algorithm learning on the base of labeled data, i.e. the data used to train the model includes both input features and the corresponding desired output, which allow it to learn over time[40].
In fact the purpose of this research is trying to replicate the behaviour of the driver drowsiness predictor algorithm (PredictS), property of Sleep Advice Technologies Srl, that classifies the level of awareness of the driver on the base of rKSS scale [41], into 3 final levels.
Indeed this can be defined a supervised learning data mining classification problem, which plans to be handled by recognizing specific patterns within the dataset and drawing some conclusions on how they should be labeled or defined.

In this research I'll rely on previous studies which defined concept of sensitivity, specificity, precision and accuracy, adapted to the topic of the problem I'm affording.
I will consider as:

- True positive (TP): predictions of the NN that coincides with the actual PredictS output, which are drowsiness states (2-3).
- True negative (TN): predictions of the NN that coincides with the actual PredictS output, which are AWAKE states (1)
- False positive (FP): predictions of the NN that are different to the actual PredictS output, which are drowsiness states (2-3).
- False negative (FN): predictions of the NN that are different to the actual PredictS output, which are AWAKE states (1)

The performances of the model will evaluated as [42]:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.1}$$

Accuracy: the proportion of the total number of correct predictions that were actually correct.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.2}$$

Positive Predictive Value or Precision: it explains how many of the correctly predicted cases actually turned out to be positive. Precision is useful in the cases where False Positive is a higher concern than False Negatives.

$$\text{Sensitivity} = \frac{TP}{TP + FN} \tag{3.3}$$

Sensitivity or Recall: the proportion of actual positive cases which are correctly identified. Recall is a useful metric in cases where False Negative is of higher concern than False Positive.

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{3.4}$$

Specificity: the proportion of actual negative cases which are correctly identified.

Precision and sensibility are the most important metrics, since this problem is characterized by the unbalancing in the data between the labels, where the awake state (level 1) dominates, and drowsy states (levels 2 and 3) are underrepresented.

All the indexes will be interpreted in parallel with graphical plots, in such a way to study the effectiveness in predicting higher levels of drowsiness, without caring of the permanence in that levels, besides studying the compliance with the original algorithm the network will try to generalize.

3.2 Feed-Forward Neural Networks and Backpropagation

The first neural network I considered was the FFNN, a type of artificial neural network where information flows in one direction, from input to output. They are composed of layers of interconnected neurons.

Multilayer feed-forward neural networks are capable of handling complex patterns in data. By using sigmoid activation functions, these networks can approximate virtually any function with sufficient hidden layers. This means they can create intricate decision boundaries, allowing them to solve problems that linear models cannot [43].
The backpropagation, a method to train multilayer neural networks, which gradually adjusts the network's connections (weights) by working backward from the output was used to train the FFNN. Initially, the network's connections are set to random, small values. Then, backpropagation repeatedly processes training data, calculating the network's output for each piece of data.

To train this model I employed the Matlab *Deep Learning Toolbox*, building a FFNN with the function *patternnet()* characterized by X number of neurons in the hidden layers, *softmax* as activation function of the output layer, since we're dealing with classification problems with 3 labels (produces a vector of probabilities summing at 1).
I set the training algorithm of the hidden layer(s) as *traingd*, which corresponds to the

descending gradient. This is a simple but effective optimization algorithm for neural network training.
The cost function: *crossentropy* is particularly suitable when using *softmax* output, because it measures the discrepancy between the probability distribution predicted by the network and the true probability distribution (one-hot) [44].

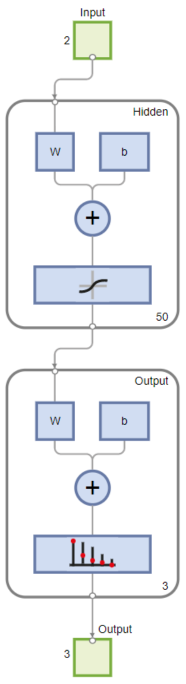


Figure 3.1: NN model structure with 1 hidden layer

The maximum number of hidden layer is keep to two because FFNN with more than two layers are not advantageous for most problems [43].

I built different FNN model, with one or 2 hidden layers [45].
If the model is fed by raw data of HR and HRV (2 features) sampled with a 1Hz frequency, the metrics evaluated on the test set (10% of the whole dataset) are really unsatisfied: approximately 70% for accuracy and specificity, but 1-2% for precision and 10-20% for sensitivity. Accuracy and specificity are not such low because of unbalancing of data between levels, in fact most of data belongs to level 1 (AWAKE): for this reason I rely mostly on precision and sensitivity.
The bad performances derive from a wrong understanding of the problem: the two features I used for training the model cannot be analyzed in a punctual way, in the sense of attributing at each sample a corresponding classification level. In fact data must be analyzed with a time-dependency trend, focusing on sections of samples, and attributing at that section the corresponding label.

The time correlation is what in FFNN is missing, even if we pre-process the data, for example taking the mean of HR and standard deviation of HRV in windows of variable size in the whole dataset.

Below the accuracy, precision, sensitivity and specificity are reported for the different sizes of the sliding window:

Window size of 10 samples: 70%-0%-0%-70%
Window size of 20 samples: 67%-4.4%-22%-70%
Window size of 30 samples: 70%-11%-31%-74%

Increasing the window to 60 samples compromises more the performances, even because we're reducing the dataset size.

At the end, this approach is useless, because even if we're creating a correlation between the samples, which are pre-processed together, this is not enough to describe the dynamical progression of the data.

3.3 Recurrent Neural Networks

Standard feed-forward neural networks are limited to analyzing static data. To process data that changes over time, we can modify these networks to incorporate information from previous moments. These enhanced networks, with their ability to consider past data, are called Recurrent Neural Networks (RNNs): they're dynamical models [46] [47]. Traditional Recurrent Neural Networks (RNNs) have difficulty processing information from distant past events (they are limited to look back for approximately 10 timesteps) [48]. This is because the signals carrying past information either weaken significantly (vanishing gradients) or intensify uncontrollably (exploding gradients) over time. To overcome this limitation, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks were developed. These networks are more adept at capturing long-term dependencies in data and can effectively process information from thousands of previous time steps [49].

LSTM and GRU units share a key similarity: both use an additive approach to update their internal state, which helps in preserving information over long sequences and addressing the vanishing gradient problem. This contrasts with traditional recurrent units that overwrite their state at each time step.
However, there are key differences: LSTM has a more complex structure with separate gates to control the flow of information into and out of the cell state. GRU is simpler, combining the forget and input gates into a single update gate. While both units have shown promising results, their relative performance can vary depending on the specific task. From these similarities and differences alone, it is difficult to conclude which types of gating units are the optimal choice for different applications [50].

I'll start from GRU, since it involves less parameters with respect to a LSTM (this factor can be relevant in terms of complexity when deploying the network on an Android application), as a consequence its training presents faster computational time, which leads the problem of tuning the hyper-parameters manually less time consuming.
Then I will exploit also LSTM functioning.

Chapter 4

Data analysis and pre-processing

4.1 Dataset: experimental activities

The dataset is made up of driving simulation, constructed on the Danisi dynamic car simulator (Torino, Italy) in September 2023 and March 2024. The merging of both experimental activities involved 24 candidates, comprising males (16) and females (8), with an average age of 33.75 years.
Each candidate was equipped with a state-of-the-art Polysomnographic device (NOX) and a Garmin Instinct 2 Dezl edition smartwatch. The smartatch was connected via Bluetooth Low Energy (BLE) to a smartphone, enabling real-time data collection through a dedicated Android application.

The subjects were supposed to drive a vehicle in a highway sunset simulated environment, with lights off inside the vehicle, following an almost straight and regular path, in such a way to relax the driver and stimulate drowsiness. The driving mission ended when the driver fell asleep or for a total driving time of approximately one hour and and half. The driver was asked to sleep less than usual the night before the experimentation, in such a way to be more exposed to possible sleep phenomena.

For both sessions, the data were analyzed by sleep expert medical doctors, to classify, for each candidate, the presence of sleep events, in compliance with the recommendations of the American Academy of Sleep Medicine (AASM).
The scored states were: non-REM stages 1,2,3, microsleep, drowsiness, walking state.
The outputs of the PredictS algorithm, were crossed with the off-line doctor scoring, and the actual feelings of the driver, to test its reliability with respect a ground truth.

The physiological parameters collected through smartwatch, i.e. the heart-rate variability (HRV) and heart rate (HR) were sampled with a 1Hz frequency and collected during the entire driving performance, associated with the on-line prediction of PredictS software.

The only difference in the set-up of the two sessions, was the alarm trilling when PredictS detected drowsiness events, turned on in Danisi 2023 and off in Danisi 2024 experimentations. The alarm related to the highest level, was set off in such a way to store data related to that drowsiness level, which is the underrepresented one during driving.

For what concerns this research, the final dataset consists of a total of 24 registrations (16 from the september session and 8 from march), collecting HR, HRV parameters and the corresponding PredictS predicted level: AWAKE, FATIGUED and DROWSY, once a second.

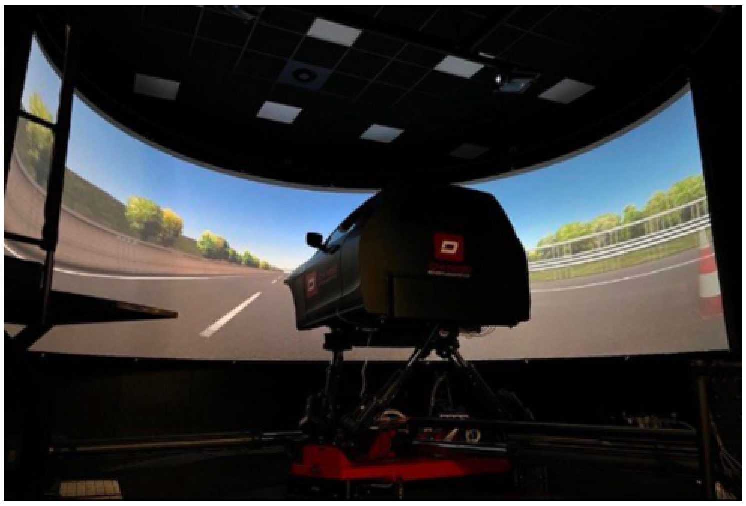


Figure 4.1: driving simulator

Figure 4.2: PSG equipment assembly

4.1.1 Data analysis and filtering

Data analysis (beside the consequent data cleaning) is an important preliminary stage, dealing with the training of a neural network. PredictS itself when analyzing the input data, leads a consistency of data check, outside the context of the computing algorithm, in such a way to avoid low quality data to interfere with the process.

When starting the PredictS algorithm some parameters HR=0 and HRV=0 are saved due to a gap in the communication between the smartwatch and the application.
Obviously these values are wrong and has to be deleted from the acquisition phase data.

In Figure 4.3 there's an example of data acquisition which at first, for few seconds, presents both the parameters sampled with a 0 value.

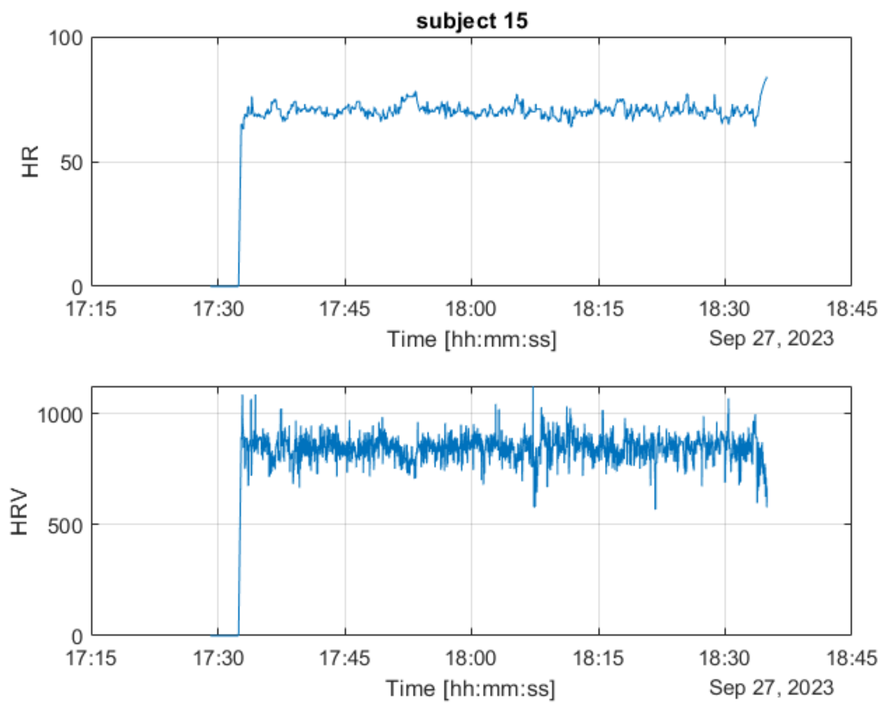


Figure 4.3: subject 15

A second aspect to focus on is the repeatability of data: due to the communication between smartphone and the cloud server, when there's a transition between states in the output of the algorithm, the very same sampled variables are reported twice, with different outputs. For this reason it's important to search for replicates and delete the copy (since data are reported with the time dependency delete the replicate related to the same time instant).

Other sources of bad quality data are found at the end of the acquisition, maybe due to the driver taking of the watch from the wrist. Also these data can be filtered, because since they are at the end of the acquisition, they will not influence the learning of the neural network. This situation can be found in 4.4.

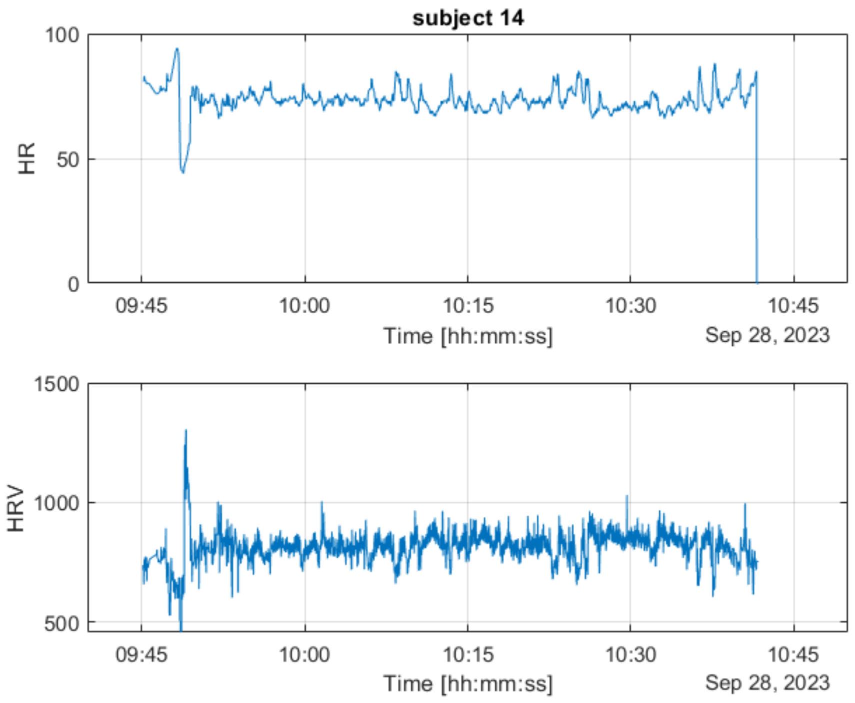


Figure 4.4: subject 14

The data cleaning approach will be different, in case low quality data are found in the middle of the experimentation, since holes in the temporal sequences can affect badly the training of the model. As we can see in Figure 4.5 the value of HRV keeps constant for more than 2 samples, that is physiologically impossible.

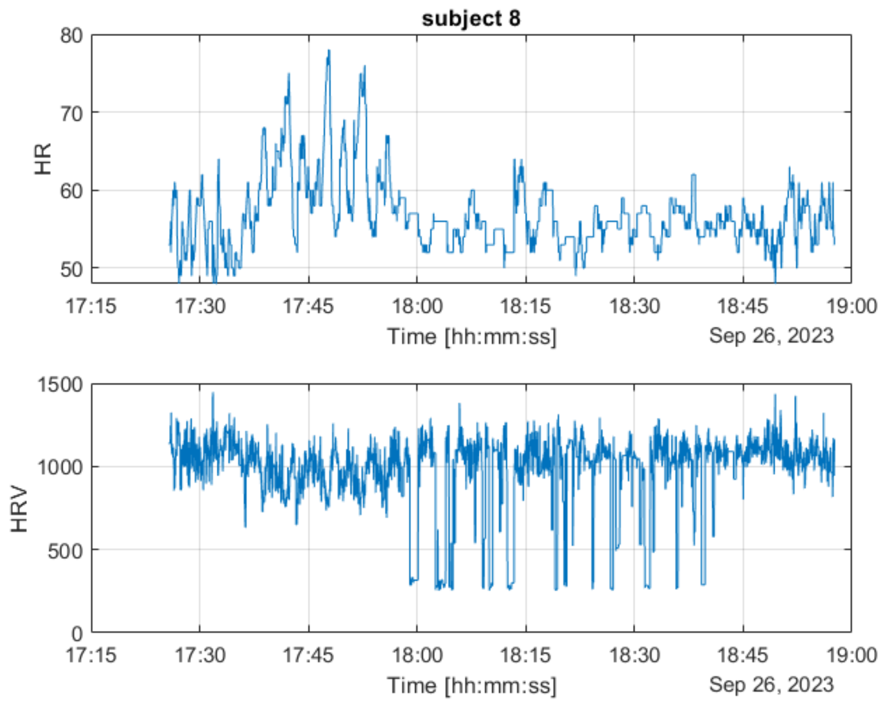
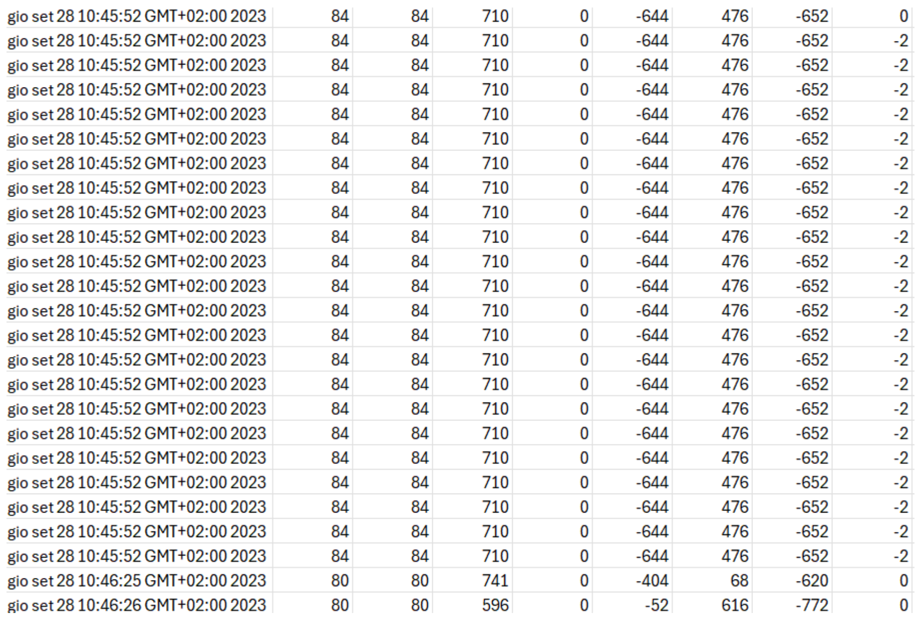


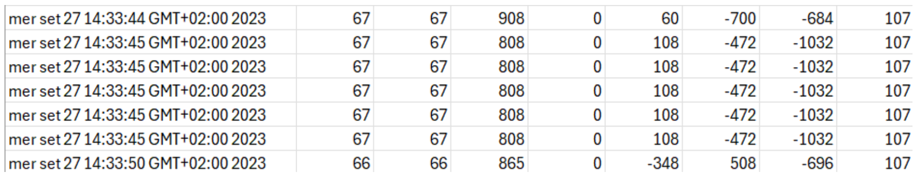
Figure 4.5: subject 8



gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	0
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:45:52 GMT+02:00 2023	84	84	710	0	-644	476	-652	-2
gio set 28 10:46:25 GMT+02:00 2023	80	80	741	0	-404	68	-620	0
gio set 28 10:46:26 GMT+02:00 2023	80	80	596	0	-52	616	-772	0

Figure 4.6: subject 5 dataset (September)

As shown in 4.6 and 4.7, the datasets collected in Danisi Engineering, especially the ones from the experimentation of September 2023, they presents holes in data due to the lack of communication with the cloud server.
In particular, the transition between a repeated sample and the successive, with the correct timing, is shown.



mer set 27 14:33:44 GMT+02:00 2023	67	67	908	0	60	-700	-684	107
mer set 27 14:33:45 GMT+02:00 2023	67	67	808	0	108	-472	-1032	107
mer set 27 14:33:45 GMT+02:00 2023	67	67	808	0	108	-472	-1032	107
mer set 27 14:33:45 GMT+02:00 2023	67	67	808	0	108	-472	-1032	107
mer set 27 14:33:45 GMT+02:00 2023	67	67	808	0	108	-472	-1032	107
mer set 27 14:33:45 GMT+02:00 2023	67	67	808	0	108	-472	-1032	107
mer set 27 14:33:50 GMT+02:00 2023	66	66	865	0	-348	508	-696	107

Figure 4.7: subject 1 dataset (September)

The approach I used for this repeated samples is filtering them, even if this would lead to intervals with missing data. In fact on the other hand, trying to re-create HR and HRV value, it could be done with linear interpolation in the case of heart-rate parameter, but the heart-rate variability is an unpredictable parameter.

4.1.2 Data processing

The python script I used in order to process data, iterates through multiple .csv files, obtained merging and cleaning .JSON files of collected data sent from the android application Predicts to the cloud server. The processing of data is done with the view of preparing training and testing datasets for the AI model, including steps of data cleaning, feature engineering, normalization, and windowing.

Function to organize input data into sliding windows:

At first I defined a function (reported below), which is in charge of transform each subject registration in a structure compatible with the RNN requests: it takes as input the matrix of collected physiological data, at the sampling frequency of 1Hz, which has as columns as the number of features interesting for the training of the NN (in my case HR and HRV) and the corresponding vector of outputs of the Predicts algorithm.

```python
def df_to_X_y(df,target,window_size):
    X=[]; Y=[]
    for i in range(len(df)-window_size):
        row = [r for r in df[i:i+window_size]]
        X.append(row)
        label=target[i+window_size]
        Y.append(label)
    return np.array(X), np.array(Y)
```

This function allows to divide the whole dataset in sliding windows of length *(window_size x number_features)* and associate, for each i_{th} window, the corresponding output located at the position *window_size + i*. In this view the obtained outputs are: a matrix *((len(df) - window_size) x window_size x number_features)*, and the corresponding vector of predicted outputs, with length *(len(df) - window_size)*, both suitable for the accomplish the structural needs of the LSTM or GRU neural network.

Initialization:

Outside the loop I defined as names the list of file names of people involved in the data collection, which took place with the support of Danisi driving simulator: each dataset related to the single subject contains the sampling 1Hz of the heart rate (HR) and heart rate variability (HRV) indices, with the corresponding acquisition time, for a total of approximately 1.5 hour of simulated driving.
I decided to merge both the data collection sessions of September 2023 and March 2024, such that the neural network can be trained on different datasets collected from different people. I had access to 16 datasets collected from 16 different people experimentation's from September 2023 and 8 datasets from March 2024, where the people examined were different from the ones of September.
Exploiting many examples coming from people with various ages, genders, and other physiological characteristics, can be crucial to allow the model to generalize on new unseen datasets, but we'll see that the quantity will not be as representative in generalizing in

some conditions.
After, I defined as local variable p, which counts the number of datasets which have been processed, so that the splits into training and testing datasets can be also done on the base of the total number of available datasets. I also initialized X_train, y_train, X_test and y_test as empty lists to hold training and testing data.

Loop through filenames:

For each file in the names list, the corresponding .csv file is read into a DataFrame *df*. The local variable inputs is created for stacking HR, HRV and Time columns red from the DataFrame; the variable called *targets* is set to the *DrowsinessState* column values, which are the drowsiness levels computed from the algorithm Predicts, each second, real time. Then I initialize *inputs_* and *y* as empty lists at each iteration of the for loop: their function will be to store filtered samples of the input and the output.

Remove duplicates:

As part of the pre-processing of raw data, there's the filtering of invalid data, usually characterized by even values of the index *TYPE*. The check of the correct *TYPE* is done in a for loop, the indices of incorrect values are saved and deleted from the inputs and targets arrays. This phenomenon happens when the user interacts with the app, for example when starting, pausing or stopping the acquisition.

Process targets: binary encoding

Predicted values saved in *targets* are copied in a new vector, called *y*, based on their values:

- if targets[k] contains bit 0 (2^0): *calibration* phase, i.e. data acquisition, which takes more or less 300 samples (5 minutes). In my code I associated to the calibration phase the 0 value
- if targets[k] contains bit 1 (2^1) or bit 2 (2^2): *awake* state, i.e. Predicts algorithm, on the base of previous samples, associated the state of attention while driving to highest level.
- if *targets[k]* contains bit 3 (2^3): *fatigued* state, i.e. Predicts algorithm, on the base of previous samples, associated the state of drowsiness while driving as intermediate level.
- if *targets[k]* contains bit 4 (2^4): *drowsy* state, i.e. Predicts algorithm, on the base of previous samples, associated the state of drowsiness while driving as high level. That means that It's predicting some minutes earlier a possible a shot sleep.
- if *targets[k]* contains bit 5 (2^5): *off-wrist*, i.e the smartwatch is far from the wrist (badly worn or not worn) and PPG signals are not collected correctly.
- if *targets[k]* contains bit 6 (2^6) or bit 7 (2^7): *low quality data*. The previous valid sample output is notified on the graphic interface.

For *targets[k]* of type *off-wrist* and *low quality data* I decided to replace them with the first valid previous level y[k-1]: this situation coincides with the real user-interface scenario: the user when dealing with poor quality data doesn't notice the changing, in the sense that the previous label keeps to be displayed and is saved in the .log file.
In this way I obtained a 3 labels classification problem.

Once modified the targets vector, with values more suitable for further computations, I set the first value of*inputs_* and *y* arrays to the first value of targets and inputs, respectively, but only if the check on consistency of HR and HRV is met.
Then I proceeded implementing a check of data repeatability, due to the communication between computing device and cloud server. In fact, when there's a change of drowsiness state, what I could see looking at data, the very same sampling is reported twice, with same input values, but different classification level; excluding the second, repeated, sample I guarantee no risk to associate the input features, to the wrong drowsiness level.
At this point I decided to delete the calibration label (0), associating data acquisition samples to the awake state (1), because I assumed people to be as alert possible, during the assembly procedure of the equipment, in view of the experimentation.
The filtered samples are then saved in *inputs_* and *y* arrays.

One-hot encoding: a technique used to represent categorical data as binary vectors, with all entries set to 0 except for the one which index is the one related to the label, set to 1. In my code one-hot encoding is used to convert the *targets[k]* values in the *targets_filtered* array, into a format suitable for multi-class classification.
Since I have no 0 level, I initialized *one_hot_targets* as an array of 0 entries of dimensions len(*inputs_filtered*) x 3 (assuming there're 3 classes based on the context) and set to each row the corresponding int(*targets_filtered[k]*)-1 entry to 1, in a for loop.
One-hot encoding is essential in machine learning, especially for categorical target variables: by converting categories into a binary format, the algorithms can process and learn from the data effectively. In my neural network, the output layer will use *softmax* activation function, for multi-class classification, which requires the target labels to be in this format, so that the network can compute the probability of each class and compare it against the one-hot encoded target during training to calculate the loss and update the weights.

Sequence of WINDOW_SIZE_SHORT lenght:

For the purposes of method 2 (I'll describe later) I decided to pre-process the whole dataset dividing it into sliding windows of length WINDOW_SIZE_SHORT samples, corresponding to 10 seconds of registration, and for each interval calculate the standard deviation of the HRV and the mean of the HR values.

Final windows: sequence of samples of WINDOW_SIZE_LONG lenght

The obtained array of pre-processed input data of dimension *(len(inputs_filtered)-WINDOW_SIZE_SHORT, num_feature)* or an array of row data of lenght *(total_dataset_lenght x num_features)*, depending on which method I'm referring to:

method 2 or 1 respectively, is then processed through a sliding window of length WINDOW_SIZE_LONG, generating larger windows *X_long* and *Y_long*. I tried three different sizes of the sliding windows to compare their differences: 180, 240 and 300, which corresponds to periods of around 3, 4 and 5 minutes respectively, since data are sampled with a frequency of 1Hz (1 sample/sec).

Split data:

1. Method A: based on the attempt of gather all the temporal windows, from each dataset and split them in the train, validation and test set in such a way for guaranteeing the percentage of each label presence based on the size of the set, using the 70%-20%-10% division. This will be better explained in the subsection *Handle unbalanced data*

2. Method B: In order to train the neural network I decided to consider ideally each single dataset as 4% (24/100) of the total, and in this view to establish the 75-12.5-12.5% division into training, validation and test sets respectively, I put in the training list the first 21 datasets (18 for training and 3 for validation: they will be successively split during the cross-validation). On the other hand, the last 3 datasets will be used to test the model's ability to generalize on new unseen data.
   I've chosen subject 22 (M), 23 (G), and 24 (V) datasets from Danisi 24 because they present variability in the predicted outputs. I avoided to change at each iteration of cross-validation the test set, to reduce the number of possible combinations.
   This represents an ideal situation, since each registration doesn't have the exact same number of samples: there could be missing data due to the miscommunication with the server or invalid ones consequence of a wrong lecture from the smartwatch. For this reason, and for the different representation of the three levels in each dataset at each iteration, at the end of each training of the different folds the performances will be different.

3. Method C: I also opted for another method: a division 19 for training, 3 for validation and 2 for testing, utilizing subject 23 (G) and 24 (V) datasets for testing, which aligns as a 80%-10%-10% division looking at the lengths of each dataset.

4.1.3 Z-score normalization of data

Outside the for loop, I made a normalization of the training and test datasets while maintaining the structure of the temporal windows. I started by concatenating all members of the list *X_train*, which is a list of numpy arrays, each representing a batch of samples with dimensions *(tot_samples, WINDOW_SIZE_LONG,num_features)*, in an array *X_train_combined*, which gather all temporal windows in all datasets used for training. The dimension of the array will be *(total_samples, WINDOW_SIZE_LONG,num_features)*, where *total_samples* is the sum of samples across all batches in *X_train*.
Then I proceeded calculating the mean and standard deviation across the combined training data: these statistics are computed along the first two axes (0,1), that means the calculation considers all samples and all-time steps within each sample but separately for each feature (HR and HRV). The results are two values for each feature: mean and std for HR and HRV respectively.
At this point I can normalize the training data: each batch of X_train is standardized using the calculated mean and standard deviation. This is done maintaining the original structure of the temporal windows.
The normalization formula applied is (x – mean_train)/std_train, where x is each batch in *X_train*.
The next step is to normalize the test data *X_test*, using the same statistics calculated from the training data (*mean_train* and *std_train*). This ensures that the normalization

applied to the test data is consistent with the training data normalization, which is crucial for maintaining model performance.

```python
def normalize_data(X_train, X_test):
    # concatenate arrays in the list X_train
    X_train_combined = np.concatenate(X_train, axis=0) # dimension (tot_samples, WINDOW_SIZE_LONG, 2)

    # obtain mean and standard deviation on combined training data
    mean_train = np.mean(X_train_combined, axis=(0, 1)) #mean along the first 2 dimensions
    std_train = np.std(X_train_combined, axis=(0, 1)) #std along the first 2 dimensions
    # normalize training data maintaining the original temporal window structure
    X_train_norm = [(x - mean_train) / std_train for x in X_train]
    # normalize testing and validation data maintaining the same statistics of the training set
    X_test_norm = (X_test - mean_train) / std_train
    X_val_norm = (X_val - mean_train) / std_train

    return X_train_norm, X_test_norm, X_val_norm
```

Even if, making a normalization on each person dataset could be effective in finding features proper of the considered set of data, I've anyway chosen to normalize each dataset with respect to the statistics (mean and standard deviation) of the whole training portion of data, so that real-time sampling normalization will be coherent with the one used during the training of the model.

4.1.4 Scheduler function

I decided to set as learning rate of the optimizer an exponential decrescent, with decay rate 0.1, which leads the learning rate to decrease of 10% of the original one at each epoch (I've also tried 0.05 as decay rate, which refers to a 5% decrement at each epoch, but this method is a little less stable than the chosen one, in terms of diminishing of the validation loss). The behavior is described in the following function, which takes as inputs the current epoch and the current learning rate, both passed automatically by the Keras training loop.

```python
def scheduler(epoch, lr):
  initial_lr = 0.01
  min_lr = 0.001
  decay_rate = 0.1

  new_lr = initial_lr *tf.math.exp(-decay_rate*epoch)

  if new_lr<min_lr:
    new_lr=min_lr
  return float(new_lr)

# Define the callback for the variable learning rate
lr_callback = tf.keras.callbacks.LearningRateScheduler(scheduler)
```

This function saturates the learning rate to a minimum value of 0.001, to prevent it from getting too small. Usually talking about NN the interval between 0.01 and 0.001 is one of the best for setting the learning rate: 0.01 is often used at the start of training to make rapid progress in minimizing the loss and 0.001 is a common default rate for many optimization algorithms, small enough to ensure stable training and good convergence properties. It was a good compromise in the view of reduce the updating of weights during training, leading to find a good minimum of the loss function and don't cause oscillations in the training accuracy. After defining the scheduler function, I created a built-in Keras callback that allows to specify how the learning rate should be adjusted at each epoch. It takes a function as an argument (in our case is the scheduler).

4.1.5 Cross-validation method

Cross-validation is a procedure used to evaluate machine learning models: the original set is divided into training and test set. The model is trained with the training set and its performances are evaluated on the test set, but this is done multiple times, to obtain a robust model.
I've already split the original available experimentation from 24 different people into 21 (for training and validation) and 3 (for testing), which means approximately 87.5%-12.5% of the overall (ideally all datasets are long approximately the same).
I decided to analyze two different types of cross-validation procedures to the training set.

The first approach consists on apply the k-fold cross-validation method taking the 90% of the total 21 dataset (even if they don't have the same number of samples) for training the model and the rest 10% for evaluate the performances (validation set). In proportion 19 for training and 2 for validating (Method C).
As a matter of fact, it could be considered a 10-fold problem, but since the validation set involves the combination of 2 different datasets I preferred to analyze all possible combinations of this subset. For this reason I wrote the code below, where at first I divided into two arrays the index related to males and females, to be sure that the validation set would involve for sure both genders, to guarantee variability. Another idea could be merging the datasets taking into account the age of subjects, but unfortunately almost everyone is in the 20-40 years age range, with a prevalence in 20-30, which is not so representative in terms of age differences.
I decided to use for the validation only the samples from Danisi 23 to make shorter the cross-validation process.
With the command itertools.combinations() I generated the combinations of couples of indices, which are successively filtered to ensure that each sequence contains one male and one female subject. At the end the command random.shuffle() mixes the combination randomly.

```
import itertools
import random
#to find all combinations of 2 from 2 vectors:
sequences=[]
array_M=np.array([0,2,3,4,5,6,7,9,13]); #male subjects
array_F=np.array([1,8,10,11,12]) #female subjects
# concatenate the two vectors
combined_array = np.concatenate((array_M, array_F))
```

```python

# obtain all combinations with 2 elements
combinations_of_2 = list(itertools.combinations(combined_array , 2))

# filter the combinations to guarantee that they contains one element
    from each vector
valid_combinations = [
    combo for combo in combinations_of_2
    if any(elem in array_M for elem in combo) and any(elem in array_F
   for elem in combo)
]
# merge the combinations in random way
random.shuffle(valid_combinations)
# save in a list to be accessible for successive computations
sequences = [combo for combo in valid_combinations]
#total of 45 sequences
```

Finally, the shuffled valid combinations are stored in the list *sequenze*, that's used for each trial of the training, with different combination of the validation set.

The second approach was the classical 75%-12.5% division of the training dataset (Method B), as I suggested previously, that means using 18 subsets for training and 3 for validating the model at each iteration (with the same procedure of creating the sequences of 3 datasets to vary the validation set). The problem is that the number of combinations to test increased up to 270, and It requires a bigger computational effort.

```python
sequences=[]
array_M=np.array([0,2,3,4,5,6,7,9,13]); #male
array_F=np.array([1,8,10,11,12]) #female
# concatenate the 2 vectors
combined_array = np.concatenate((array_M, array_F))

# find all combinations of 3 elements
combinations_of_3 = list(itertools.combinations(combined_array , 3))

# filter the combinations to guarantee che ci siano 2 elementi da
    entrambi i vettori
valid_combinations = [
    combo for combo in combinations_of_3
    if any(elem in array_M for elem in combo) and any(elem in array_F
   for elem in combo)
]
# merge the combinations in random way
random.shuffle(valid_combinations)
# save in a list
sequences = [combo for combo in valid_combinations]
#total of 270 sequences
```

Below the indication of the index related to each dataset:

0	subject1 27-09-2023	12	subject13 29-09-2023
1	subject2 29-09-2023	13	subject14 27-09-2023
2	subject3 28-09-2023	14	subject15 28-09-2023
3	subject4 28-09-2023	15	subject16 28-09-2023
4	subject5 28-09-2023	16	subject17 13-03-2024
5	subject6 28-09-2023	17	subject18 13-03-2024
6	subject7 27-09-2023	18	subject19 12-03-2024
7	subject8 26-09-2023	19	subject20 12-03-2024
8	subject9 27-09-2023	20	subject21 13-03-2024
9	subject10 27-09-2023	21	subject22 12-03-2024
10	subject11 28-09-2023	22	subject23 13-03-2024
11	subject12 29-09-2023	23	subject24 12-03-2024

Chapter 5

GRU model

GRU is a specialized recurrent neural network (RNN) architecture widely used in deep learning. Unlike standard feedforward networks, GRUs incorporate feedback connections, allowing them to capture temporal dependencies in sequential data.
Similarly to LSTMs, they are specifically designed to overcome the vanishing and exploding gradient problems often encountered when training traditional RNNs on long data sequences, making them ideal for tasks like natural language processing, speech recognition, and time series forecasting. Compared to LSTMs, GRUs have a simpler structure and fewer parameters, making them easier to train and more computationally efficient.
Unlike LSTMs, which use separate memory and hidden states, GRUs combine these into a single hidden state, updated using two gates: the **reset gate** and the **update gate**. The reset gate controls how much of the previous hidden state to forget, while the update gate determines how much of the new information (candidate activation) is incorporated into the next hidden state.
GRUs are preferred when simpler architectures or limited computational resources are required. For this reason it is the class of RNNs I wanted to start employing to face this classification problem.

```
model=Sequential()
    model.add(InputLayer((WINDOW_SIZE_LONG , 2)))
    model.add(GRU(64,activation='tanh',
        recurrent_activation='sigmoid',
        kernel_initializer=GlorotUniform(),
        recurrent_initializer=Orthogonal()))
    model.add(Dense(3,'softmax'))
```

Above the lines of code to build the neural network model:

- Input gate: Sequences of shape (*WINDOW_SIZE* timesteps, 2 features).
- GRU LAYER: a Gated Recurrent Unit layer is added with 64 units.

I opted for this type of recurrent neural network (RNN) because it's particularly effective for handling sequential data, addressing the vanishing gradient problem.
I've chosen an hyperbolic tangent as the activation function for the output of the GRU units and a sigmoid function as activation function for the recurrent step.
The hyperbolic tangent (tanh) activation function is typically chosen for the output of GRU units because it provides a good balance between allowing both positive and negative

outputs (ranging from -1 to 1). This property makes it effective in capturing patterns in sequential data. It also helps to mitigate the vanishing gradient problem by maintaining some gradient flow during backpropagation, thus preserving the long-term dependencies in sequences.
Sigmoid activation is used within the gates of the GRU (update and reset gates) because it squashes values between 0 and 1, which can represent probabilities or gating mechanisms. This allows the network to "decide" how much of the past information to retain or discard, a key aspect of GRU's effectiveness in handling long sequences. The sigmoid function is well-suited here as it provides smooth control over how much information flows through the network. [51]

Furthermore, for recurrent neural networks such as Gated Recurrent Units (GRU), weight initialization is equally crucial to ensure good convergence during training (the risk is incurring in diminishing of the training accuracy through epochs).
For this reason a good practice is to use the GlorotUniform initializer [52] (also known as Xavier), a method which works well with both linear and non-linear activations and is often used for recurring networks.
Kernel_initializer is a parameter used to initialize the weights of the GRU input layers, while *recurrent_initializer* initializes the weights of recurrent connections and It's frequently used in RNN to increase stability in long-term sequencies.
GlorotUniform and *Orthogonal* are the methods set for the two parameters to improve the stability and the convergence of the model.
GlorotUniform: balances input and output variances, which is important to prevent gradient vanishing and exploding. *Orthogonal* maintains the orthogonality of recurring matrices, which helps keep information along sequences, useful for problems with long time dependencies. [53]
I've tried to add a dropout 0.1 to increase the validation accuracy, but in this case it prevented the model to learn correctly from training data, leading to a diminishing of the training accuracy, so I deleted it.
I've also tried to add a 100 layer Dense layer to increase the complexity and increase the model's capacity to learn more complex patterns, but this doesn't improve the performances.

- DENSE LAYER: at the end a dense (fully connected) layer *Dense(3)* with 3 units is added to the model.

This layer will output a 3-dimensional vector, whose activation function: *softmax* is used to normalize the output so that it represents a probability distribution over 3 classes.

Layer (type)	**Output Shape**	**Param #**
GRU (gru_50)	(None, 64)	13,056
Dense (dense_50)	(None, 3)	195

Total params:	13,251
Trainable params:	13,251
Non-trainable params:	0

5.1 Method B and C: training with cross-validation

The following lines of code perform the cross-validation method to train and evaluate a GRU (Gated Recurrent Unit) deep neural network model for sequential data classification. Each fold involves defining the model, training it, evaluating its performances, and saving the model. The explanation follows:

```
person_age=[24,28,63,27,29,75,32,35,44,30,33,34,29,52,29,31]
person_gender=[0,1,0,0,0,0,0,0,1,0,1,1,1,0,0,0]
for fold_index in range(0,270):
    print(f"Fold {fold_index+1}:")
    X_fold_train = [] ; y_fold_train = []
    indici_val=np.array(sequenze[fold_index])
    print(indici_val)
    X_val=X_train[indici_val[0]]; y_val=y_train[indici_val[0]]

    for i in indici_val[1:]:
      X_val = np.vstack((X_val, X_train[i]))
      y_val = np.vstack((y_val, y_train[i]))
    print(X_val.shape)

    # Exclude validation set from training
    for i in range(len(X_train)):
      if i not in indici_val:
          X_fold_train.append(X_train[i])
          y_fold_train.append(y_train[i])

    # Convert lists to numpy arrays
    X_fold_train =  np.vstack(X_fold_train)
    y_fold_train =  np.vstack(y_fold_train)

    X_fold_train = np.array(X_fold_train, dtype=float)
    y_fold_train = np.array(y_fold_train, dtype=float)
    X_val = np.array(X_val, dtype=float)
    y_val = np.array(y_val, dtype=float)

    print(X_fold_train.shape,y_fold_train.shape)
    print(X_val.shape,y_val.shape)

    # Calculates the mean and standard deviation over all data of
   training combined
    mean_train = np.mean(X_fold_train, axis=(0, 1))  # Average over
   the first two dimensions
    std_train = np.std(X_fold_train, axis=(0, 1))     # Deviation
   standard along the first two dimensions
    print("Mean:", mean_train)
    print("Standard Deviation:", std_train)

    X_train_norm= (X_fold_train - mean_train) / std_train
    X_test_norm = (X_test - mean_train) / std_train
    X_val_norm = (X_val - mean_train) / std_train

    del modelGRU_cross
    modelGRU_cross=Sequential()
    modelGRU_cross.add(InputLayer((WINDOW_SIZE_LONG, 2)))
    modelGRU_cross.add(GRU(64,activation='tanh',
        recurrent_activation='sigmoid',
        kernel_initializer=GlorotUniform(),
        recurrent_initializer=Orthogonal()))
  # modelGRU_cross.add(Dense(100, activation='relu',
 kernel_initializer=HeNormal()))
  modelGRU_cross.add(Dense(3,'softmax'))

  modelGRU_cross.compile(loss='categorical_crossentropy',
 optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
  metrics=['accuracy'])

  # Train the model with early stopping based on validation loss
  early_stopping = EarlyStopping(monitor='val_loss', patience=6,
 restore_best_weights=True)
  modelGRU_cross.fit(X_train_norm, y_fold_train, epochs=50,
 validation_data=(X_val_norm, y_val),
 callbacks=[early_stopping,lr_callback], batch_size=32,
 shuffle=True) #aggiunto shuffle=True

  # Evaluate the model on the validation set
  loss, accuracy = modelGRU_cross.evaluate(X_val_norm, y_val)
  print(f'Validation Loss: {loss}, Validation Accuracy: {accuracy}')

  # Evaluate the model on test data
  loss, accuracy = modelGRU_cross.evaluate(X_test_norm, y_test)
  print(f'Test Loss: {loss}, Test Accuracy: {accuracy}')

  from keras.models import save_model
  modelGRU_cross.save(f"C:\\Users\\catia\\
  fold{fold_index+1}.keras")
```

1. Data preparation:
   Person_age and *person_gender* are lists containing ages and genders of people involved in the experimentations, as the order they've in the vector names. Having this information helped me to divide the people's dataset into two arrays to discriminate the gender, some passages before, to got the sequenze array. *X_train_norm* and *y_train* are normalized training data and corresponding labels, while *X_test_norm* and *y_test* are normalized test data and corresponding labels.

2. Cross-validation loop:
   The loop iterates from 0 to 270, indicating the different folds for cross-validation. For each fold, the indices of the *X_train_norm* list that will be concatenated to fill the validation set are retrieved from sequenze, iteratively. On the other hand, the indices which exclude the ones related to the validation, are deployed to fill the training set.

3. Model definition:
   For each fold the existing model instance is deleted to ensure that each fold starts with a new model and a new sequential model is then created. An input layer is added to the model: the input shape is specified as *(WINDOW_SIZE_LONG , 2)*, indicating that the model expects sequences of length WINDOW_SIZE_LONG and 2 features per timestep.
   GRU LAYER: explaned previously.

4. Compilation of the model:
   The model is compiled with the loss function *categorical cross-entropy*, used for multi-class classification and the Adam optimizer with a specific learning rate. The metric used to evaluate the model's performance during training and evaluation is based on accuracy. *X_fold_train* and *y_fold_train* are the training data and labels. The maximum number of training epochs is set to 50, but it's never reached because the training stops before, due to the *early_stopping* callback. Then validation data are reported (both inputs and outputs) and the list of callbacks is defined: *early_stopping* and *lr_callback*.

5. Training:
   The model is trained using the *fit* method with *early stopping*, to prevent over-fitting by monitoring the validation loss and stopping when it stops improving. Then it saves the model with the lowest validation loss.

6. Evaluation on the experimental dataset:
   After training the model is evaluated on both the validation set and the test set. This provides an assessment on the model's performance.

7. Evaluation on data collected with different procedures:
   I had access on additional data coming from real driving scenario, which stands apart from experimental controlled behaviour, in order to understand if, even with a small experimental dataset used for training the model, the resulting model is able to generalize on data collected with other procedures.

8. Saving the model:
   The trained model for each fold is saved to a file

5.2 Method B and C: training achievements

5.2.1 Window size of 300 and row input data

The whole dataset composed of 24 registration was spitted in 19 registrations for training, 3 for validation and **2 for test** following the percentages 80%-10%-10%.

At first I decided to test the performance of the GRU model in training with row data, i.e. HR and HRV values, that fill a sliding window of 300 samples, which scrolls of 1 sample every second.

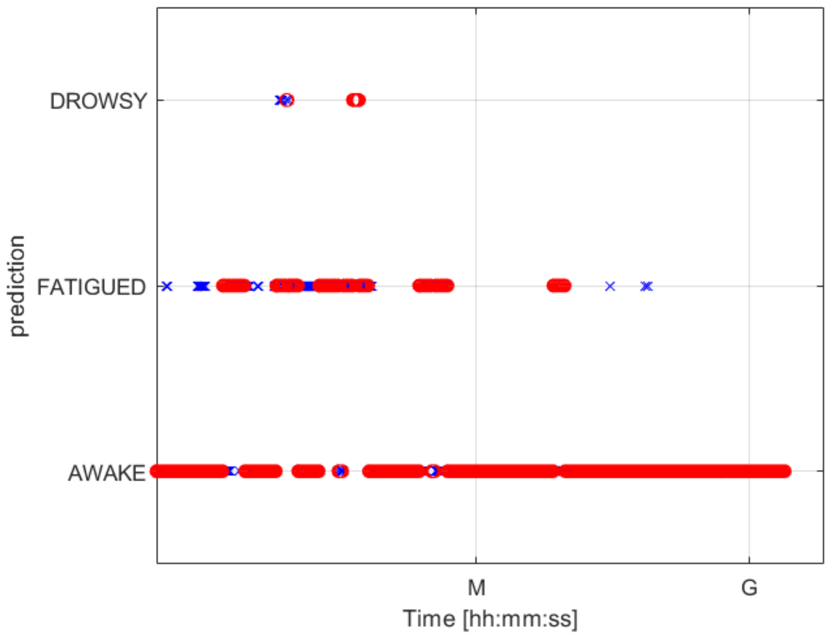


Figure 5.1: fold 8: test set ('ro': PredictS, 'bx': GRU)

I obtained the following metrics: Accuracy: 0.873, Precision: 0.66, Sensitivity 0.74, Specificity 0.906.

In 5.1 the red circles are the outputs of PredictS algorithm, the blue crosses are the predictions of the GRU model.

I'll always use this color-symbol convention in the following plots.
The train set in each fold is made up by gathering of data collected from the experimentation of Danisi 2024, by (in order) G (subject 23) and V (subject 24).
I've chosen these datasets because they allow to obtain a quite balanced test set with respect drowsy levels (2 and 3), which are the less represented: in fact G ones presents FATIGUED and DROWSY episodes, that are necessary in test set in order to evaluate the predictions around these labels; V was picked as the second test dataset in order to not deprive the training and validation datasets of representative examples of higher drowsiness levels prediction.

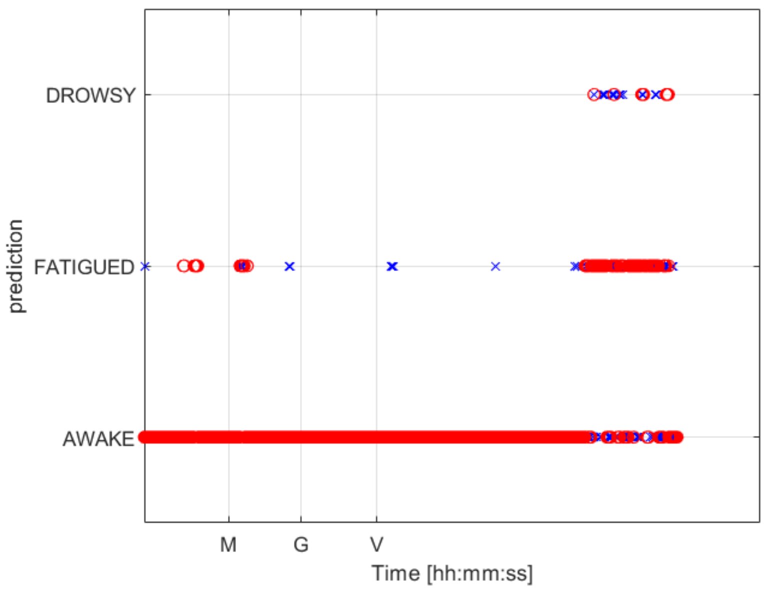


Figure 5.2: fold 8: MV-highway ('ro': PredictS, 'bx': GRU)

Accuracy: 0.93, Precision: 0.782, Sensitivity 0.681, Specificity 0.97

I had access to additional datasets collected during everyday activities by subjects non involved in the experimentation took in Danisi Engineering, allowing to evaluate the performances of the model in real scenario. The registration displayed above was made by the user during an highway travel. The behaviour is approximately coherent with PredictS software and this is important also because the subject didn't participate to the experiments (his dataset is effective as test set). Additionally, the user is 53 years old, age which is far from the one prevalent in the training set, polarized around 30.

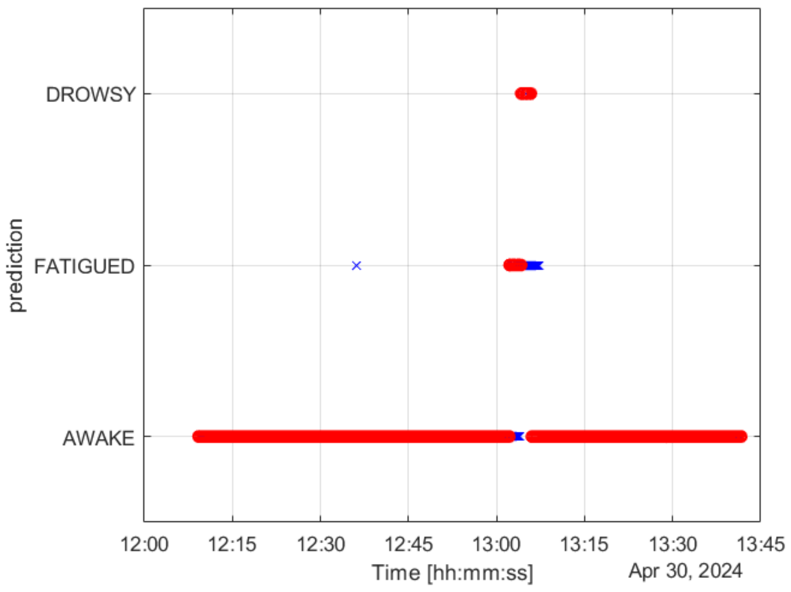


Figure 5.3: fold 8: MV-lunch ('ro': PredictS, 'bx': GRU)

Accuracy: 0.97, Precision: 0.657, Sensitivity 0.626, Specificity 0.986

The registration analyzed above was made on the same subject, during lunch time. As we can see from the graph, the GRU replicates the same behaviour of PredictS predictions in transictioning from AWAKE to FATIGUED states, even if this is a 'bug' in PredictS logic.

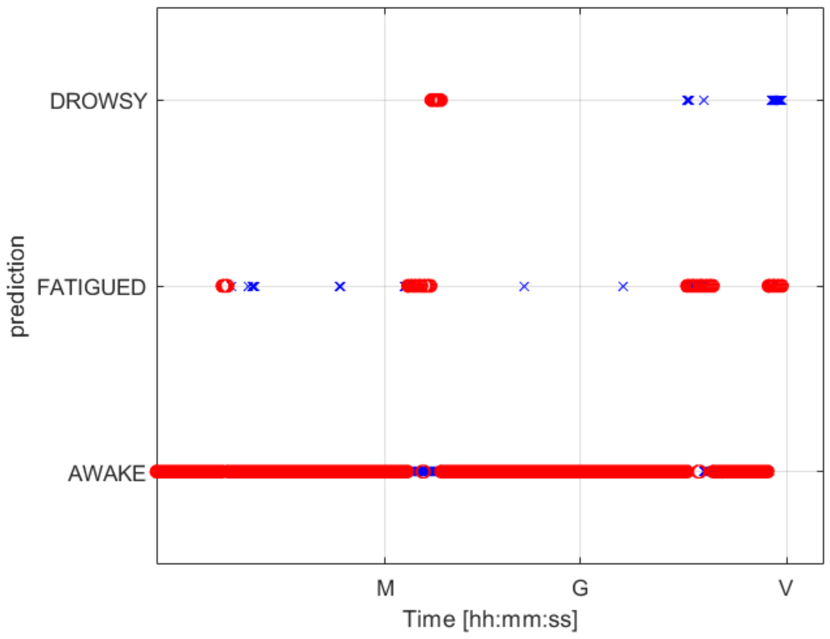


Figure 5.4: fold 8: RG20240504 ('ro': PredictS, 'bx': GRU)

As shown in 5.4, both the FATIGUED and DROWSY levels are delayed with respect to the true predictions. The model doesn't perform well.
Also in *rg_20240501.csv* dataset, with a precision of 0.31 and a sensitivity of 0.28, the overall behaviour of the model predictions seems too polarized on FATIGUED state, with a disperse trend with respect to the true one.

5.2.2 Window size of 300 and row input data - filtering of temporal windows

The whole dataset composed of 24 registration was spitted in 19 registrations for training, 3 for validation and **2 for test** following the percentages 80%-10%-10%.

In this method I've filled the buffer of 10 samples at first, then checked if there're more than 2 samples where the value of HRV replicates. If yes, that temporal window is removed and scrolls by one sample, on the other hand if the check gives a negative feedback the algorithm computes the mean of the 10 values of HR and the standard deviation of the HRV, with the related label level in output and these are used to fill a 300 steps temporal window.

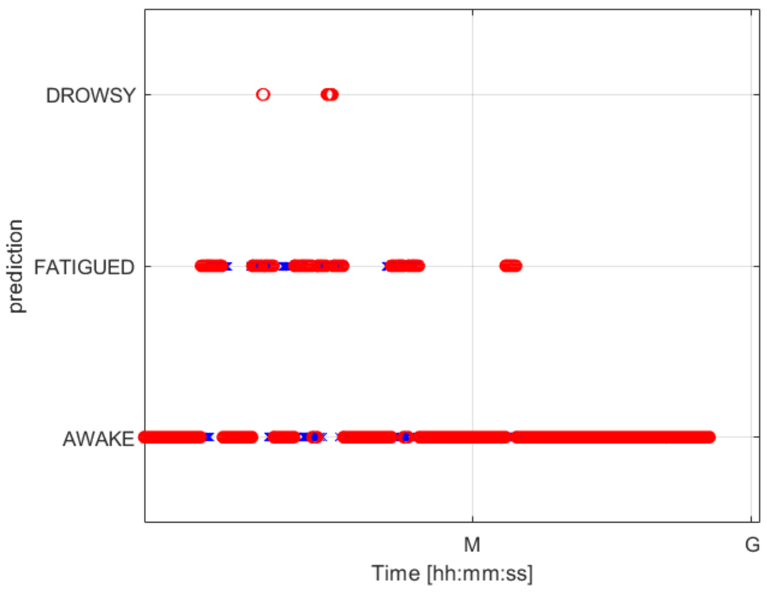


Figure 5.5: fold 17: test set ('ro': PredictS, 'bx': GRU)

Accuracy: 0.884, Precision: 0.811, Sensitivity: 0.609, Specificity: 0.961

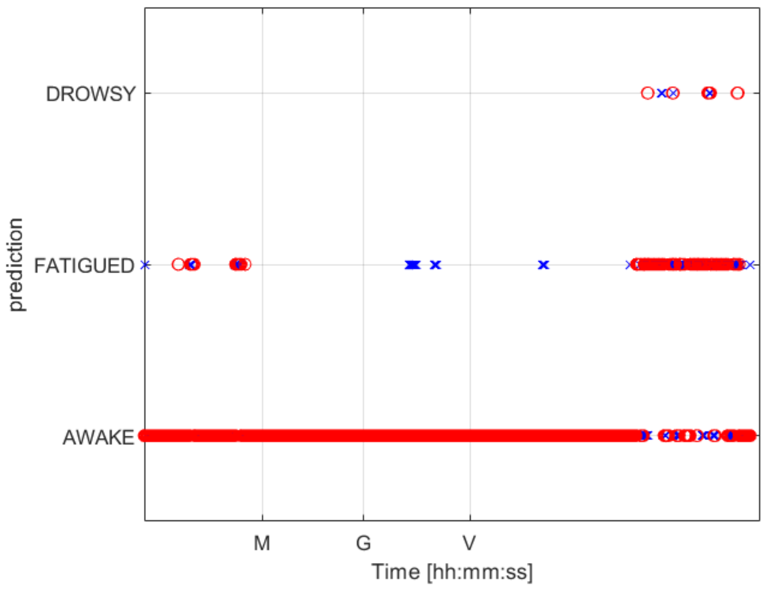


Figure 5.6: fold 17: MV-highway ('ro': PredictS, 'bx': GRU)

Accuracy: 0.9198, Precision: 0.698, Sensitivity: 0.787, Specificity: 0.94

About the registration rg_20240430: 0.97%-..-..0.97% (there's only the AWAKE level)

5.2.3 Window size of 300 and pre-processed input data

The whole dataset composed of 24 registration was spitted in 18 registrations for training, 3 for validation and **3 for test** following the percentages 87.5%-12.5% (training+validation - testing).
Below, the code for pre-processing the data used to train the model (batch size=32). In this first trial I haven't filtered the training data from low quality data, even if in the PredictS application, a procedure of data-cleaning is performed, before injecting the data in the algorithm.
I only deleted repeated samples, due to the lack in the communication with the cloud server.
This represents the most realistic scenario, where the user every second receives the level of attention, even if the data are corrupted (for low quality data the output displayed in correspondence of low quality data is the one of the previous sample. The only notification is a button on the interface, which from green becomes gray)

The following metrics are related to the binary problem: AWAKE vs FATIGUED-DROWSY. But my reasoning I mostly relied on the graphical observation, considering (for example) as true positive also predictions happening in the boundary of the true label, not in a punctual way.

3 datasets in validation									
fold	validation sequence	train acc	train loss	val acc	val loss	test acc	test prec	test Sens	test spec
9	[3 9 1]	0.937	0.159	0.8	0.678	0.825	0.63	0.54	0.91
12	[2 7 11]	0.875	0.298	0.885	0.31	0.83	0.71	0.4	0.95
13	[13 8 10]	0.85	0.34	0.8	0.56	0.823	0.894	0.24	0.99
14	[6 1 12]	0.877	0.29	0.737	0.637	0.869	0.74	0.64	0.94
18	[0 13 10]	0.878	0.287	0.782	0.55	0.848	0.826	0.42	0.975
20	[4 13 11]	0.89	0.26	0.64	1.16	0.81	0.56	0.62	0.86
21	[5 1 8]	0.87	0.307	0.764	0.518	0.86	0.78	0.52	0.96
23	[0 2 10]	0.83	0.38	0.85	0.329	0.886	0.825	0.63	0.96
24	[2 1 10]	0.886	0.268	0.719	0.722	0.82	0.68	0.39	0.95
25	[5 9 10]	0.87	0.31	0.876	0.267	0.84	0.77	0.43	0.96
27	[5 7 11]	0.865	0.31	0.85	0.356	0.84	0.65	0.64	0.9
29	[3 4 8]	0.896	0.25	0.796	0.729	0.85	0.72	0.54	0.94
32	[5 8 10]	0.87	0.3	0.84	0.42	0.83	0.7	0.45	0.94
33	[4 13 8]	0.835	0.375	0.85	0.32	0.85	0.86	0.4	0.98
35	[5 7 12]	0.84	0.375	0.85	0.32	0.83	0.82	0.3	0.98
38	[0 7 8]	0.878	0.29	0.844	0.44	0.85	0.71	0.57	0.93
40	[0 7 1]	0.893	0.253	0.8	0.65	0.88	0.84	0.55	0.97
42	[3 5 11]	0.905	0.21	0.79	0.656	0.84	0.898	0.3	0.99
43	[0 4 10]	0.88	0.27	0.72	0.73	0.8	0.63	0.33	0.94
44	[3 4 12]	0.82	0.35	0.71	0.75	0.78	0.52	0.45	0.87
45	[0 2 8]	0.88	0.274	0.75	0.65	0.82	0.67	0.36	0.95
51	[2 3 12]	0.9	0.2	0.78	0.51	0.8	0.7	0.22	0.97
60	[7 13 8]	0.944	0.165	0.86	0.35	0.83	0.65	0.52	0.92
72	[7 8 11]	0.9	0.21	0.78	0.54	0.8	0.7	0.22	0.97
80	[3 4 1]	0.924	0.18	0.7	0.73	0.86	0.81	0.5	0.97

For improving the performances in the testing datasets I added 2 conditions, to the code written above, in order to filter the data:

```
***(1): cont=0
***(2): and input_row[0]>=30
***(3): if input_row[1]==inputs[i-1,1]:
              cont+=1
            else:
              cont=0
            if cont>=2: #3 HRV successivi uguali
              for j in range(2):
                inputs_filtered.pop()
                targets_filtered.pop()
              cont=0
```

In (1) a local variable cont is initialized to 0: it will be a counter for the repetition of the same HRV value in consecutive sampling.
Thanks to (2) condition, I consider only plausible value of HR values, that in non-pathological condition usually doesn't go below 30 bpm (I reasoned about the possibility for the subject to be bradycardic).
This helps to filter possible low quality data due to the communication between smartwatch and smartphone.
The (3) condition instead checks if HRV is constant through successive samples, and if it keeps the same value for more than 3 consecutive values, they're not considered and the counter is reset to 0.
This condition is in charge of filtering corrupted data, in such a way to get the dataset closer to the one injected in PredictS algorithm.
In some datasets it helps to reach the DROWSY state.
In the following pages when there's () symbol it means that the filter is applied, if () is missing it means that data are not filtered.

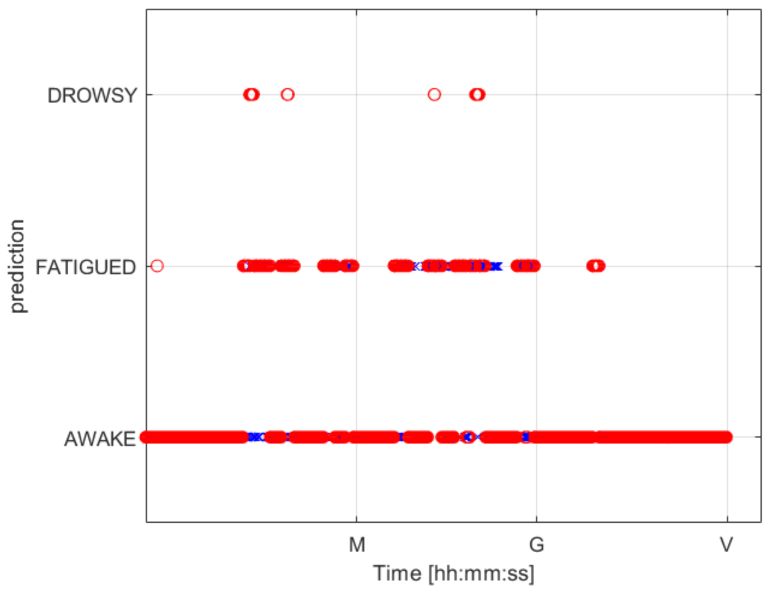


Figure 5.7: test set-fold 80 () ('ro': PredictS, 'bx': GRU)

Accuracy:0.88 , Precision: 0.82 , Sensibility: 0.62 , Sensitivity: 0.96

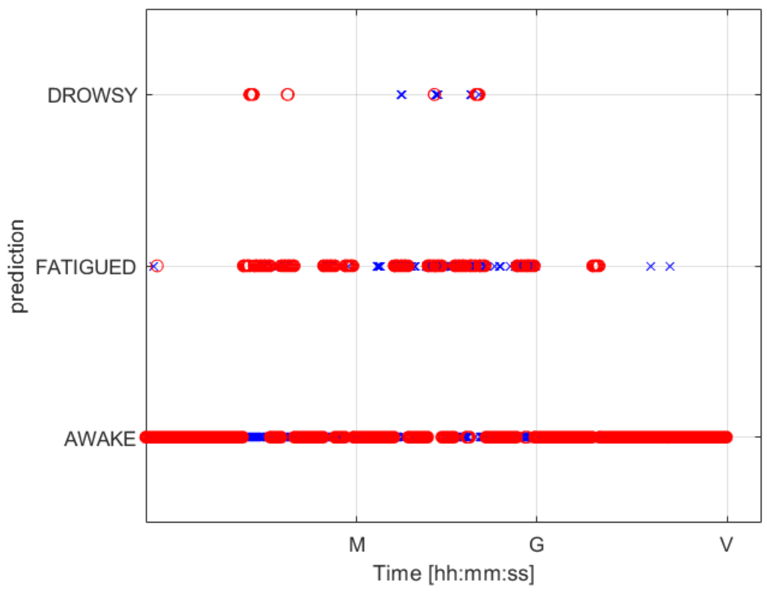


Figure 5.8: test set-fold 12 () ('ro': PredictS, 'bx': GRU)

Accuracy: 0.809 , Precision: 0.686 , Sensitivity: 0.376 , Specificity 0.946

The train set in each fold is made up by gathering of data collected from the experimentation of Danisi 2024, by (in order) M (subject 22), Gea (subject 23) and V (subject 24). I've chosen these datasets because allow to obtain a quite balanced test set with respect drowsy levels (1 and 2), which are the less represented: in fact M and G ones presents

FATIGUED and DROWSY episodes, that are necessary in test set in order to evaluate the predictions around these labels.

Fold 80: other test sets from non-experimental scenario

I had access to additional datasets collected during everyday activities by subjects non involved in the experimentation took in Danisi Engineering, allowing to evaluate the performances of the model in real scenario.

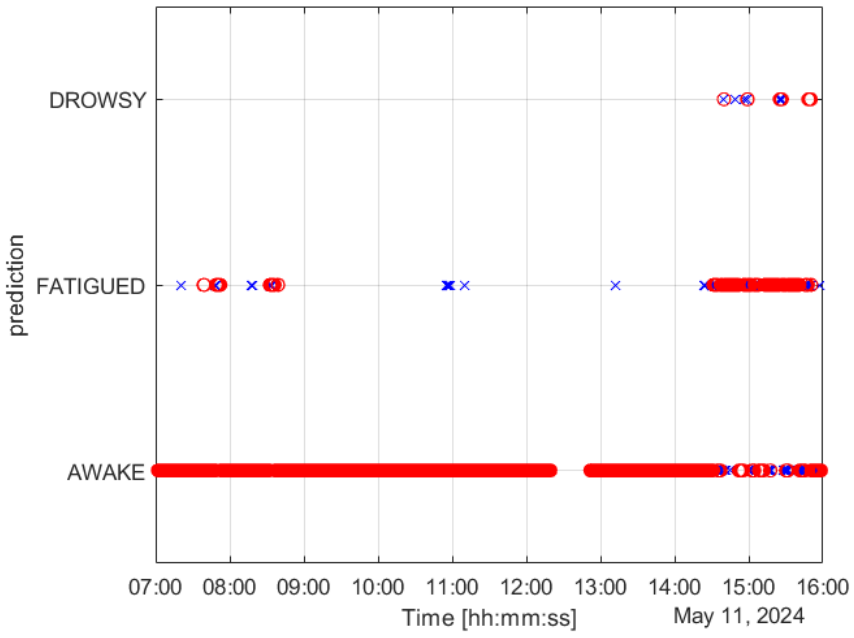


Figure 5.9: fold 80: MV-20240511 ('ro': PredictS, 'bx': GRU)

The registration displayed above was made by the user during an highway travel. The behaviour is approximately coherent with PredictS software and this is important also because the subject didn't participate to the experiments (his dataset is effective as test set). Additionally, the user is 53 years old, age which is far from the one prevalent in the training set, polarized around 30.

Accuracy: 0.924 , Precision: 0.73 , Sensitivity: 0.76 , Specificity: 0.96

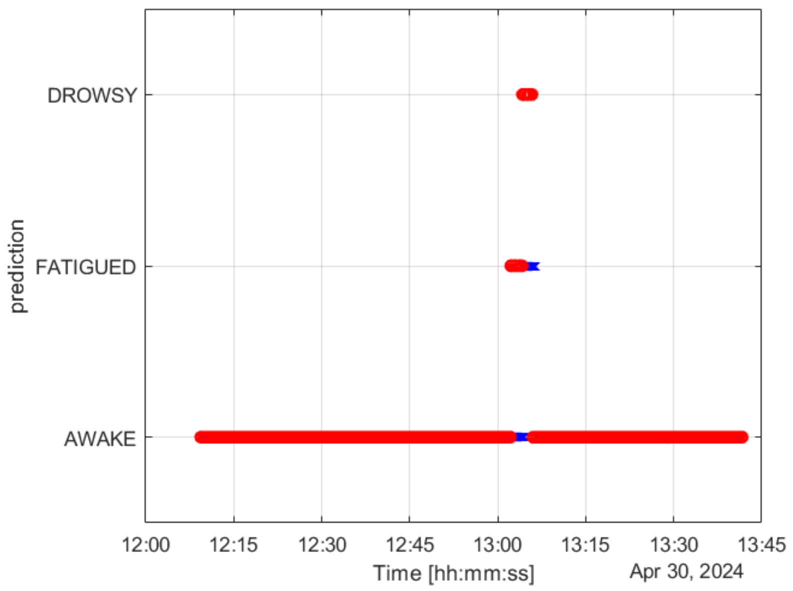


Figure 5.10: fold 80: MV-lunch ('ro': PredictS, 'bx': GRU)

This dataset was collected during lunch time and replicates an inconsistency in the outputs of PredictS, because the level 2 and 3 (FATIGUED and DROWSY) shouldn't be present and are caused by a train of invalid data. A filtering of the input data will solve the problem, also in the sw case. As it's shown in the picture above, the FATIGUED state is reached by the NN prediction almost 2 minutes after the sw one, while the DROWSY is not.

Accuracy: 0.97, Precision: 0.75, Sensitivity: 0.4, Specificity: 0.99

Filtering the input data, deleting HR values greater than 0 and HRV which replicate, leads to the very same behaviour of the output. Notice that since some samples are deleted, the FATIGUED level is not maintained for the whole interval: the NN replicates the very same behaviour but with some minutes latency.

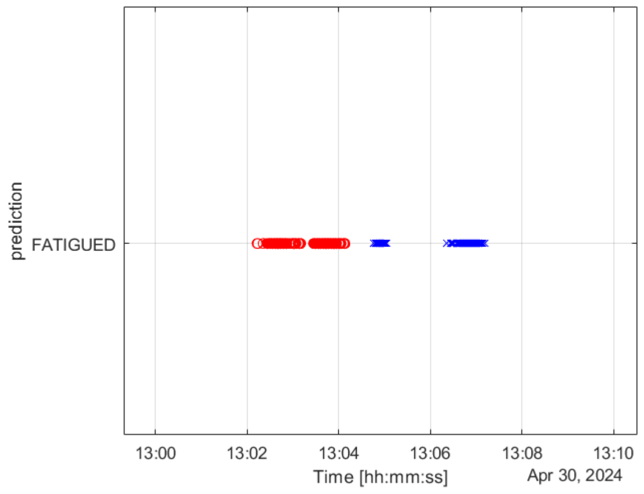


Figure 5.11: fold 80 (): MV-lunch ('ro': PredictS, 'bx': GRU)

In the next page, the model obtained with the fold 80 is used to predict the labels of some registrations took by an user, who is 60 years old and has participated to the experimentation of Danisi 2023. The following results are important in order to generalize the performances of the NN to ages not correlated with the ones prevalent in the training.

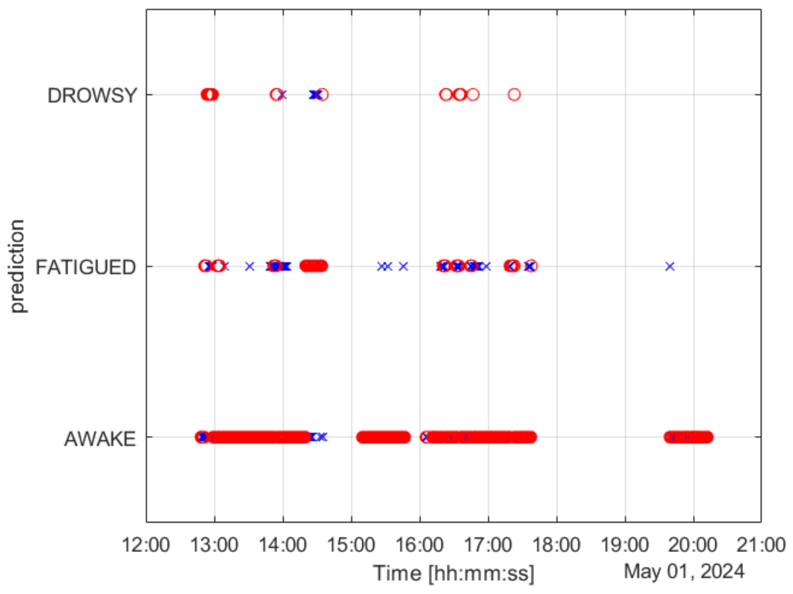


Figure 5.12: RG-20240501() ('ro': PredictS, 'bx': GRU)

Accuracy: 0.829, Precision: 0.49, Sensitivity: 0.443, Specificity: 0.907

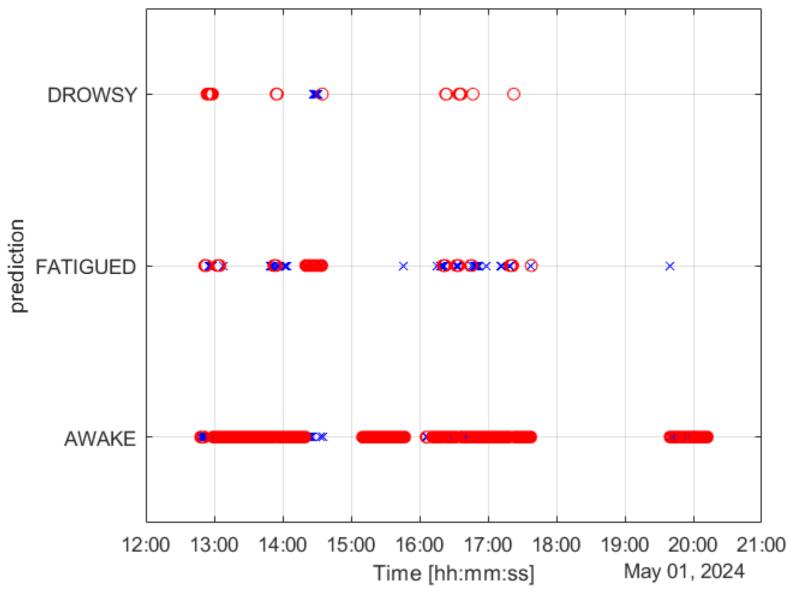


Figure 5.13: RG-20240501 ('ro': PredictS, 'bx': GRU)

Accuracy: 0.85, Precision: 0.55, Sensitivity: 0.429, Specificity: 0.932

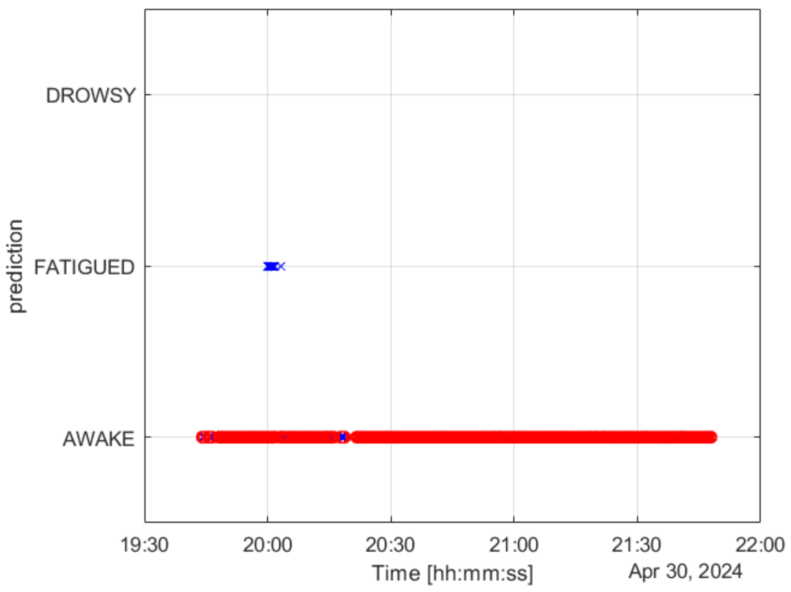


Figure 5.14: RG-20240430() ('ro': PredictS, 'bx': GRU)

Accuracy: 0.985 , Specificity: 0.985

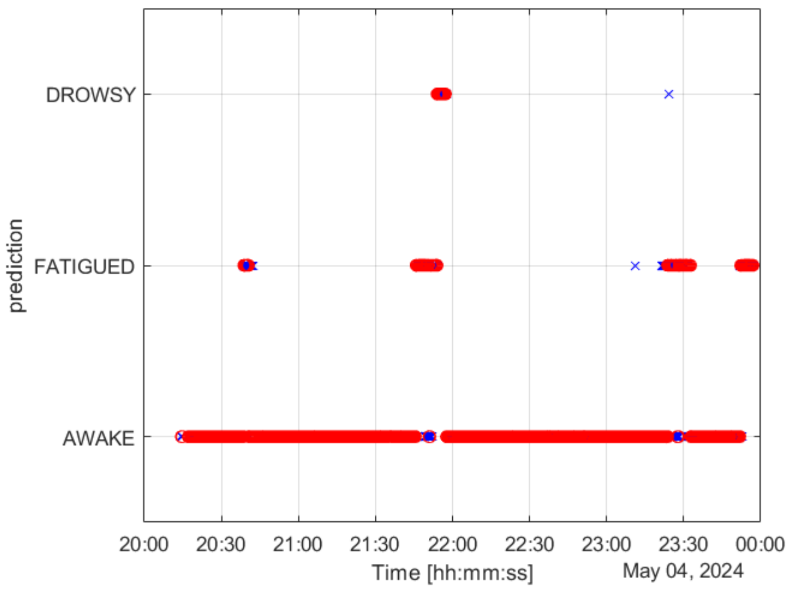


Figure 5.15: RG-20240504() ('ro': PredictS, 'bx': GRU)

Accuracy: 0.944, Precision: 0.81, Sensitivity: 0.714, Specificity: 0.976 (all levels)

Trials with larger batch sizes

When using a very large batch size results in more precise gradient estimates because the gradient is computed over more data points, reducing the stochastic noise in the gradient estimation. While this precision might seem beneficial, it can actually harm the model's ability to generalize to new data: with less noise, the optimizer is more likely to converge to sharp minima in the loss landscape, which usually correspond to solutions that fit the training data very well but do not perform well on unseen data.
Smaller batch sizes introduce more noise into the gradient estimates, that can help the optimizer to escape sharp minima and instead find flatter minima, which are often associated with better generalization performance.

Furthermore, when using optimizers like Stochastic Gradient Descent (SGD), a large batch size decreases the variance in the gradient estimates: with less variance, the optimizer may take larger steps (updates), which can cause it to overshoot the minimum or oscillate around it, making the optimization process less stable. This can lead to difficulty in convergence and potentially cause the optimizer to diverge.

Another aspect to considerate is that large batch sizes provide a very precise gradient estimate that may not have enough "noise" to push the optimizer out of the plateaus, which are regions where the gradient is very small. As a result, the optimizer might take a long time to escape plateaus or get stuck in local minima, leading to slower convergence. The inherent noise in smaller batch sizes can help the optimizer to jump out of plateaus by providing occasional larger gradient estimates, which can give the model the push needed to continue making progress.

Batch size of 64

Choosing a batch size of 64 leads almost to the same performances of batch size of 32 in the test set.

Batch size of 128

Choosing a batch-size of 128 leads to a slightly deterioration of the performances in the test set (output almost never reach the DROWSY state).

At this point I made some other trials in training the GRU model utilizing 2 registration as validation data, in such a way to reduce the maximum number of combinations to cover the entire cross-validation method to 45. I opted for different lenght of the sliding windows used to pre-process the data.

In the tables below are reported the most relevant results when training the GRU model with registrations of sequence *fold* used as validation set.
I didn't report all the models for sake of simplicity.

5.2.4 Window size of 300 and pre-processed input data - validation with 2 datasets

fold	validation sequence	train acc	train loss	val acc	val loss	test acc	test prec	test Sens	test spec
2	[2 12]	0.88	0.27	0.79	0.42	0.84	0.68	0.63	0.91
11	[3 8]	0.93	0.17	0.85	0.39	0.84	0.73	0.47	0.95
22	[9 12]	0.87	0.3	0.88	0.33	0.83	0.68	0.46	0.93
27	[2 8]	0.96	0.097	0.9	0.28	0.84	0.73	0.5	0.94
33	[6 12]	0.95	0.11	0.84	0.46	0.87	0.7	0.7	0.9

5.2.5 Window size of 240 and pre-processed input data - validation with 2 datasets

fold	validation sequence	train acc	train loss	val acc	val loss	test acc	test prec	test Sens	test spec
1	[2 10]	0.97	0.087	0.86	0.48	0.87	0.76	0.67	0.93
4	[7 1]	0.93	0.17	0.78	0.52	0.83	0.8	0.37	0.97
6	[9 8]	0.84	0.36	0.94	0.15	0.81	0.81	0.25	0.98
8	[6 8]	0.84	0.36	0.92	0.21	0.82	0.68	0.44	0.94
9	[0 1]	0.89	0.25	0.73	0.78	0.86	0.76	0.6	0.94
11	[3 8]	0.9	0.24	0.86	0.38	0.85	0.67	0.75	0.89
12	[5 10]	0.9	0.25	0.84	0.37	0.84	0.87	0.32	0.99
15	[9 10]	0.88	0.28	0.89	0.19	0.88	0.75	0.73	0.92
16	[0 12]	0.9	0.24	0.78	0.57	0.86	0.74	0.64	0.93
19	[5 1]	0.9	0.21	0.71	0.78	0.84	0.8	0.4	0.97
22	[9 12]	0.87	0.31	0.88	0.31	0.86	0.76	0.56	0.94
24	[0 11]	0.86	0.31	0.82	0.41	0.83	0.8	0.35	0.97
28	[13 12]	0.91	0.2	0.66	0.67	0.85	0.67	0.66	0.9
30	[3 11]	0.86	0.32	0.81	0.53	0.86	0.87	0.45	0.98
32	[9 1]	0.85	0.33	0.82	0.47	0.84	0.75	0.46	0.95
33	[6 12]	0.85	0.34	0.83	0.4	0.8	0.84	0.2	0.99
35	[2 11]	0.95	0.12	0.85	0.44	0.84	0.73	0.4	0.96
37	[7 11]	0.85	0.34	0.85	0.37	0.84	0.68	0.5	0.93
38	[5 8]	0.86	0.32	0.84	0.32	0.84	0.7	0.48	0.94
41	[9 11]	0.87	0.29	0.88	0.33	0.84	0.67	0.54	0.92
45	[3 10]	0.85	0.33	0.80	0.58	0.86	0.88	0.43	0.98

5.2.6 Window size of 128 and pre-processed input data - validation with 2 datasets

fold	validation sequence	train acc	train loss	val acc	val loss	test acc	test prec	test Sens	test spec
2	[2 12]	0.84	0.34	0.85	0.36	0.83	0.92	0.22	0.99
3	[3 1]	0.87	0.31	0.73	0.82	0.83	0.73	0.39	0.96
4	[7 1]	0.88	0.27	0.83	0.45	0.83	0.65	0.46	0.93
6	[9 8]	0.84	0.36	0.94	0.15	0.84	0.77	0.41	0.97
7	[6 11]	0.86	0.32	0.77	0.55	0.84	0.7	0.46	0.94
11	[3 8]	0.86	0.32	0.86	0.34	0.82	0.63	0.42	0.93
12	[5 10]	0.9	0.24	0.84	0.4	0.84	0.75	0.39	0.96
19	[5 1]	0.89	0.26	0.75	0.6	0.84	0.7	0.45	0.95
22	[9 12]	0.89	0.26	0.83	0.33	0.84	0.72	0.42	0.95
23	[3 12]	0.89	0.26	0.76	0.66	0.82	0.62	0.46	0.92
24	[0 11]	0.85	0.36	0.77	0.54	0.83	0.75	0.3	0.97
25	[4 8]	0.86	0.33	0.78	0.54	0.83	0.78	0.32	0.98
36	[0 8]	0.84	0.35	0.85	0.33	0.83	0.74	0.31	0.97
39	[6 10]	0.85	0.35	0.88	0.34	0.84	0.65	0.64	0.9
43	[7 8]	0.86	0.32	0.91	0.32	0.84	0.7	0.5	0.94
45	[3 10]	0.84	0.35	0.81	0.43	0.83	0.83	0.27	0.98

I analyzed sliding windows of lenght 300, 240 and 180, which corresponds to 5,4 and 3 minutes of driving respectively.
Since in this type of problem, it's important the fact that model is able to notice the incoming of a sleep event, more than the metrics values, I analyzed graphically the behaviour of the models' outputs in each case, to find the worst average performances when training the model with windows of lenght 180.

5.3 Age as an additional feature

The whole dataset composed of 24 registration was spitted in 18 registrations for training, 3 for validation and **3 for test** following the percentages 87.5%-12.5% (training+validation - testing).

Since I obtained the best performances with the sliding window of 240 till now, I've decided to proceed with the same procedure, exploiting more additional features as the age of people involved in Danisi 2023 and 2024 experimentations.
I expect the model behaving correctly for people around 30 years old, because the prevalence of subjects in the whole training dataset is polarized in that age range, but the performances will be worse for older people.

As input data, the Danisi datasets from 2023 and 2024 were filtered in such a way to be presumably consistent to the ones processed by PredictS.
Since ideally, training data should be treated in a similar way to test data, I considered filtering the training data in the same way to avoid discrepancies.

The filtering method I used before (deleting data with HR less than 30 (presumably HR=0 due to errors in the lecture of the PPG signal from the smartwatch and samples with HRV which repeats the very same value for at least 3 sampling), seemed to behave correctly with testing data, because this filtering injects in the model data that are closer to the ones that PredictS is supposed to process.

The first approach I used for normalization of input data was normalize each feature of the training set separately, with the Z-normalization method.
The mentioned filtering will be applied on both training and testing data.

fold	validation sequence	train acc	train loss	val acc	val loss	test acc	test prec	test Sens	test spec
2		0.96	0.099	0.85	0.43	0.85	0.75	0.52	0.95
6		0.88	0.26	0.92	0.34	0.85	0.75	0.56	0.94
8		0.94	0.15	0.9	0.28	0.85	0.68	0.68	0.94
15		0.84	0.35	0.91	0.23	0.85	0.76	0.46	0.96
16		0.85	0.34	0.71	0.62	0.86	0.81	0.53	0.96
23		0.91	0.2	0.78	0.6	0.83	0.76	0.4	0.96
25		0.85	0.34	0.8	0.42	0.86	0.96	0.42	0.99

Fold 16

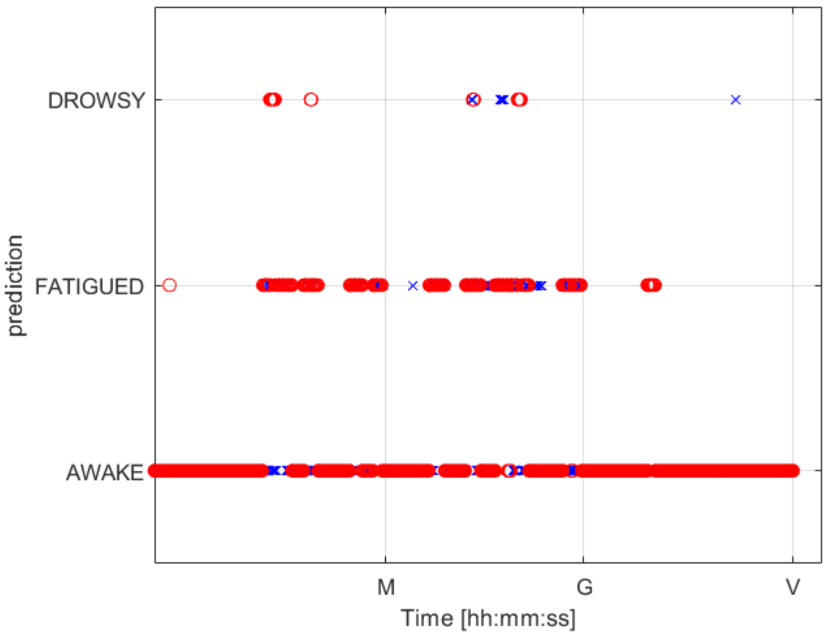


Figure 5.16: fold 16: test set ('ro': PredictS, 'bx': GRU)

The model reaches every FATIGUED state, except for the first and only one of subject V, and reaches 2 alarms for person G, where the second is dispatched 5 minutes early than the actual one. None of the alarms of M is expired by the model.

On the other hand, performances decrease considering as test data, the additional non-experimental ones: the predicted labels are more distributed on the FATIGUED level, with respect to the correct ones, causing lots of FP, while the DROWSY level is never reached.

As supposed before, the fact that the additional datasets are related to two subjects, whose age differ at least 20 years (53 and 60), from the one prevalent in the training set (32), could cause wrong behaviour of the predicted output.

Indeed, the model may not generalize well if the age in the test set is significantly different from that of the training set. This is because the model may have learned specific relationships based on age distribution in the training set that are not valid for a different distribution.

Furthermore, if the model has been trained on a data set with a certain age distribution, it may show bias towards that distribution. When tested on data with a different distribution, performance may deteriorate. This is a model bias problem. On the other hand, a model with high variance may adapt too much to the training data and fail to capture the underlying structure, leading to poor performance on test data with different distributions.

In the case of folder 25, as we can see in Fig 5.17, the distribution around the FATIGUED layer is less dense, but it never reach the DROWSY state anyway.

Another approach I used for training the model with three features, including the age, was filter the samples with HR less than 30 and delete the 10 sample windows in which

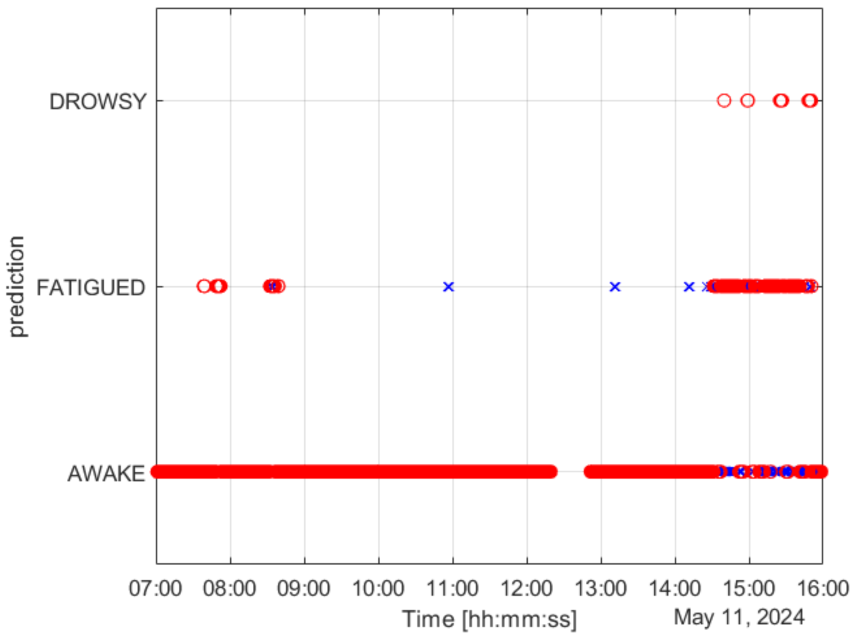


Figure 5.17: fold 25: MV-20240511 ('ro': PredictS, 'bx': GRU)

there're more or equal than 3 samples with the same value of HRV, before training the model and then use filtered data for testing; normalization of each feature by its own (window_size=240) (test parameters in binary)

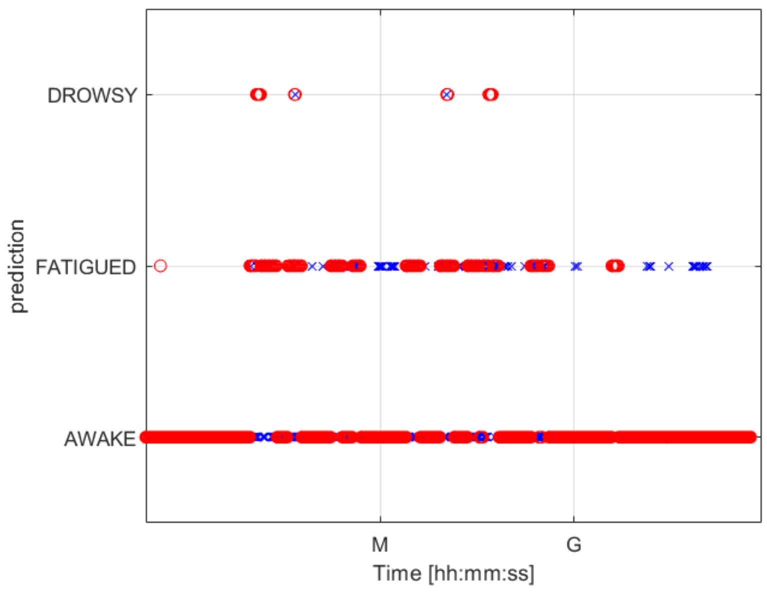


Figure 5.18: fold 44: test set ('ro': PredictS, 'bx': GRU)

Accuracy: 0.83, Precision: 0.7, Sensitivity: 0.46, Specificity: 0.94

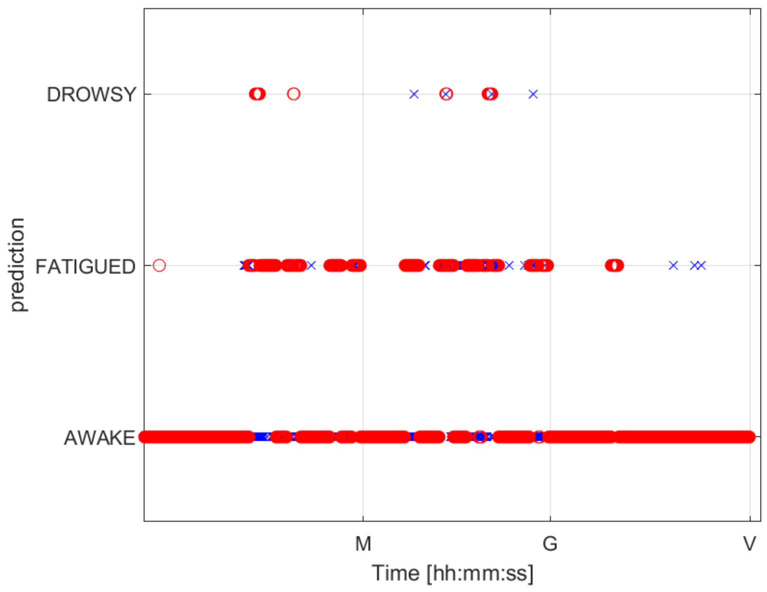


Figure 5.19: fold 40: test set ('ro': PredictS, 'bx': GRU)

Accuracy:0.84, Precision: 0.78, Sensitivity: 0.42, Specificity: 0.96

5.4 Method A: handle unbalanced data

As implicit from the nature of the application this algorithm is suitable for, i.e. detecting sleeping events during driving, the final dataset, comprehensive of all the driving registrations, presents unbalancing of data between classification labels.
The AWAKE state is the predominant in all the registrations, followed by ATTENTION (1st stage in terms of sleepiness). DROWSY state, whose presence triggers the alarm in the vehicle, is the less represented. The lack of data in the most important level to be detected, besides the overall restricted experimental dataset, represent a challenging starting point to improve the performances of the final model.
The following table shows the distribution of samples in each level for all the registrations.

subject	AWAKE	ATTENTION	DROWSY
0	2701	637	49
1	1025	1949	318
2	2031	0	0
3	1905	776	151
4	938	1015	85
5	1612	435	0
6	1143	0	0
7	2840	278	0
8	2848	484	0
9	2972	3	0
10	2265	464	0
11	2036	1153	79
12	1600	967	21
13	1476	1318	26
14	789	1329	647
15	2274	510	0
16	1945	1610	0
17	2978	91	0
18	3997	115	0
19	2560	259	0
20	2737	0	0
21	3268	1235	123
22	2415	1514	123
23	4057	161	0

Table 5.1: number of samples per level for each registration

Description of the Data Splitting Method:

1. Initial Data Concatenation: after preprocessing and windowing the data from multiple files, the processed data arrays X_300 and Y_300 from each file are appended to lists X_list and y_list, then concatenated along the first axis to form combined arrays X_combined and y_combined.

2. Class-Wise Splitting: the combined data is then split by class. Three separate lists X_1, X_2, X_3 and y_1, y_2, y_3 are created to hold data points belonging to

class 1 (AWAKE), class 2 (FATIGUED), and class 3 (DROWSY) respectively.

3. Train-Test Split: For each class-specific dataset (X_1, X_2, X_3 and their corresponding labels), a train-test split is performed using train_test_split from sklearn.model_selection. The test_size parameter is set to 0.1, meaning 10% of the data for each class is set aside for testing, while the remaining 90% is kept for training.

4. Combining Class-Specific Splits: The train and test sets from each class are concatenated to form the final training and test sets.

5. Class-Wise Splitting of Training Data for Validation: Similar to the initial split, the training data is again split by class. This is done to balance the validation set across classes.

6. Train-Validation Split: For each class-specific training dataset, a train-validation split is performed using train_test_split with the test_size parameter set to validation_size / (1 - test_size). This calculates the proportion of the original data that should go into the validation set based on the desired validation size relative to the training data. The resulting splits are concatenated to form the final training and validation set.

The splitting process ensures that each class is equally represented in the training, validation, and test sets. This is important for classification tasks to prevent bias towards a particular class.

Using train_test_split with a fixed random state ensures repeatability of the splits while maintaining randomness within each class.

```
Epoch 1/20 1934/1934 [==============================] - 581s
279ms/step - loss: 0.3145 - accuracy: 0.8730 - val_loss: 0.2317 - val_accuracy: 0.9021 -
lr: 0.0100
Epoch 2/20 1934/1934 [==============================] - 559s
289ms/step - loss: 0.1800 - accuracy: 0.9284 - val_loss: 0.1424 - val_accuracy: 0.9422 -
lr: 0.0090
Epoch 3/20 1934/1934 [==============================] - 553s
286ms/step - loss: 0.1157 - accuracy: 0.9544 - val_loss: 0.1402 - val_accuracy: 0.9491 -
lr: 0.0082
Epoch 4/20 1934/1934 [==============================] - 582s
301ms/step - loss: 0.1024 - accuracy: 0.9601 - val_loss: 0.1144 - val_accuracy: 0.9592 -
lr: 0.0074
Epoch 5/20 1934/1934 [==============================] - 518s
268ms/step - loss: 0.1270 - accuracy: 0.9500 - val_loss: 0.0773 - val_accuracy: 0.9716 -
lr: 0.0067
Epoch 6/20 1934/1934 [==============================] - 456s
236ms/step - loss: 0.0788 - accuracy: 0.9691 - val_loss: 0.0761 - val_accuracy: 0.9722 -
lr: 0.0061
Epoch 7/20 1934/1934 [==============================] - 589s
305ms/step - loss: 0.0675 - accuracy: 0.9741 - val_loss: 0.0546 - val_accuracy: 0.9781 -
lr: 0.0055
Epoch 8/20 1934/1934 [==============================] - 726s
```

```
375ms/step - loss: 0.0548 - accuracy: 0.9791 - val_loss: 0.0722 - val_accuracy: 0.9727 -
lr: 0.0050
Epoch 9/20 1934/1934 [==============================] - 558s
289ms/step - loss: 0.0580 - accuracy: 0.9783 - val_loss: 0.0410 - val_accuracy: 0.9849 -
lr: 0.0045
Epoch 10/20 1934/1934 [==============================] - 446s
231ms/step - loss: 0.0793 - accuracy: 0.9695 - val_loss: 0.0948 - val_accuracy: 0.9617 -
lr: 0.0041
Epoch 11/20 1934/1934 [==============================] - 435s
225ms/step - loss: 0.0665 - accuracy: 0.9740 - val_loss: 0.0613 - val_accuracy: 0.9758 -
lr: 0.0037
Epoch 12/20 1934/1934 [==============================] - 437s
226ms/step - loss: 0.0547 - accuracy: 0.9789 - val_loss: 0.0458 - val_accuracy: 0.9829 -
lr: 0.0033
Epoch 13/20 1934/1934 [==============================] - 451s
233ms/step - loss: 0.0449 - accuracy: 0.9827 - val_loss: 0.0523 - val_accuracy: 0.9782 -
lr: 0.0030
Epoch 14/20 1934/1934 [==============================] - 444s
229ms/step - loss: 0.0355 - accuracy: 0.9858 - val_loss: 0.0322 - val_accuracy: 0.9879 -
lr: 0.0027
Epoch 15/20 1934/1934 [==============================] - 450s
233ms/step - loss: 0.0325 - accuracy: 0.9875 - val_loss: 0.0297 - val_accuracy: 0.9891 -
lr: 0.0025
Epoch 16/20 1934/1934 [==============================] - 466s
241ms/step - loss: 0.0267 - accuracy: 0.9897 - val_loss: 0.0354 - val_accuracy: 0.9854 -
lr: 0.0022
Epoch 17/20 1934/1934 [==============================] - 452s
234ms/step - loss: 0.0258 - accuracy: 0.9899 - val_loss: 0.0258 - val_accuracy: 0.9903 -
lr: 0.0020
Epoch 18/20 1934/1934 [==============================] - 457s
237ms/step - loss: 0.0205 - accuracy: 0.9922 - val_loss: 0.0357 - val_accuracy: 0.9869 -
lr: 0.0018
Epoch 19/20 1934/1934 [==============================] - 457s
236ms/step - loss: 0.0197 - accuracy: 0.9921 - val_loss: 0.0234 - val_accuracy: 0.9916 -
lr: 0.0017
Epoch 20/20 1934/1934 [==============================] - 455s
235ms/step - loss: 0.0179 - accuracy: 0.9927 - val_loss: 0.0203 - val_accuracy: 0.9921 -
lr: 0.0015
```

Above I reported the steps of the training through each epoch. In this trial I filled the sliding windows of dimension (WINDOW_SIZE_LONG, 2) with row data (HR and HRV), sampled each second, the scrolled by 1 sample.

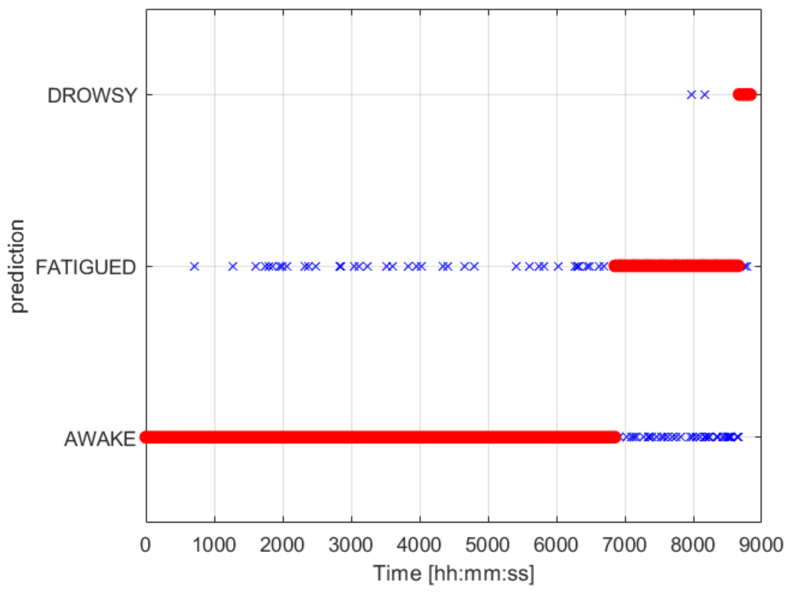


Figure 5.20: test set ('ro': PredictS, 'bx': GRU)

Test set (binary): Accuracy: 0.9899, Precision: 0.9798, Sensitivity: 0.975, Specificity: 0.994

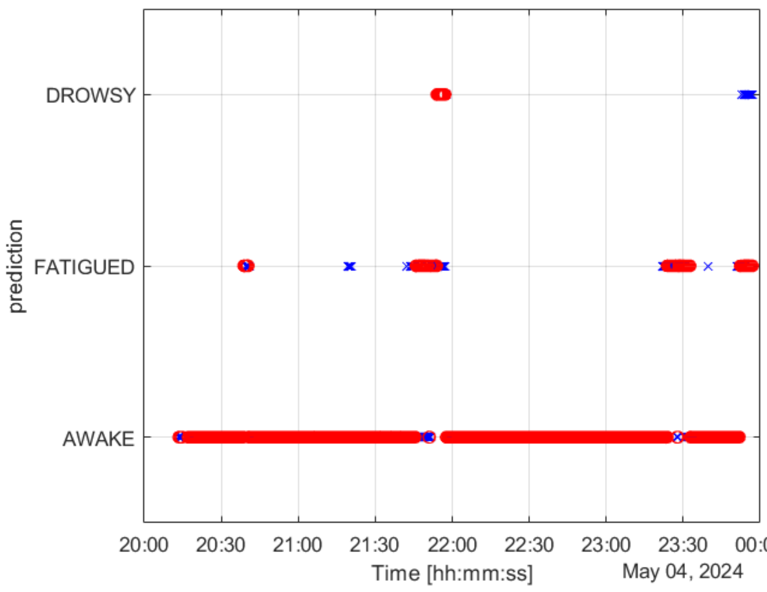


Figure 5.21: RG0405 ('ro': PredictS, 'bx': GRU)

Accuracy: 0.93, Precision: 0.742, Sensitivity 0.63, Specificity 0.97

In Fig 5.21 the output of this model (blue crosses in the graph) replicates the almost same behaviour of 5.4, proving that two methods work similarly.

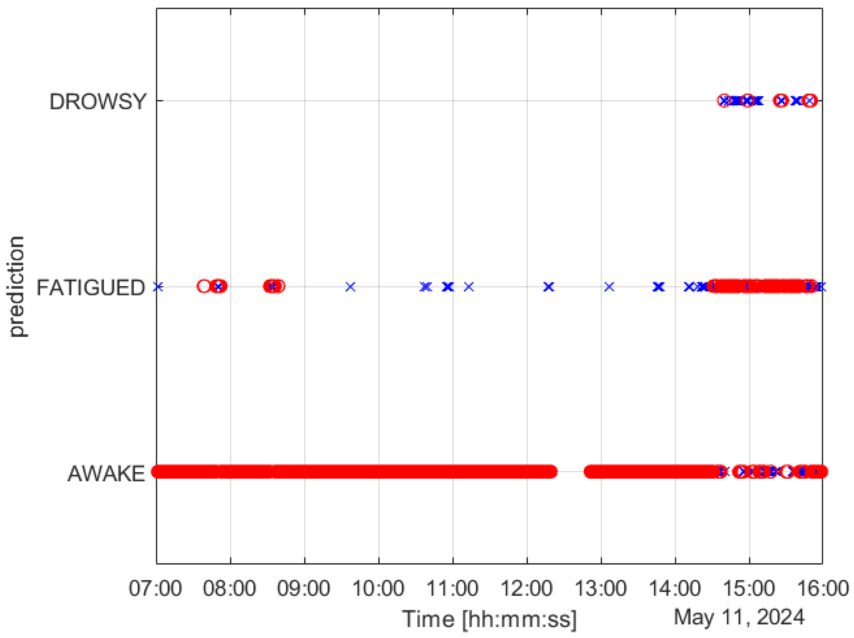


Figure 5.22: MV20240511 ('ro': PredictS, 'bx': GRU)

Accuracy: 0.9088, Precision: 0.675, Sensitivity 0.644, Specificity 0.951

In a second moment I tried to train the GRU with sliding windows of dimension *(WINDOW_SIZE_LONG, 2)* filled by computing each second, since the $WINDOW\_SIZE\_SHORT_{th}$ one, the mean of HR and standard deviation of HRV of the previous WINDOW_SIZE_SHORT samples.

The train stopped at the 10th epoch, for the *EarlyStopping* of patience 6

Epoch 10/20 1929/1929 [==============================] - 448s 232ms/step - loss: 0.0783 - accuracy: 0.9709 - val_loss: 0.0621 - val_accuracy: 0.9751 - lr: 0.0041 Test Loss: 0.06432885676622391, Test Accuracy: 0.9749347567558289

Test set: (binary) Accuracy: 0.975, Precision: 0.934, Sensitivity: 0.957, Specificity: 0.98

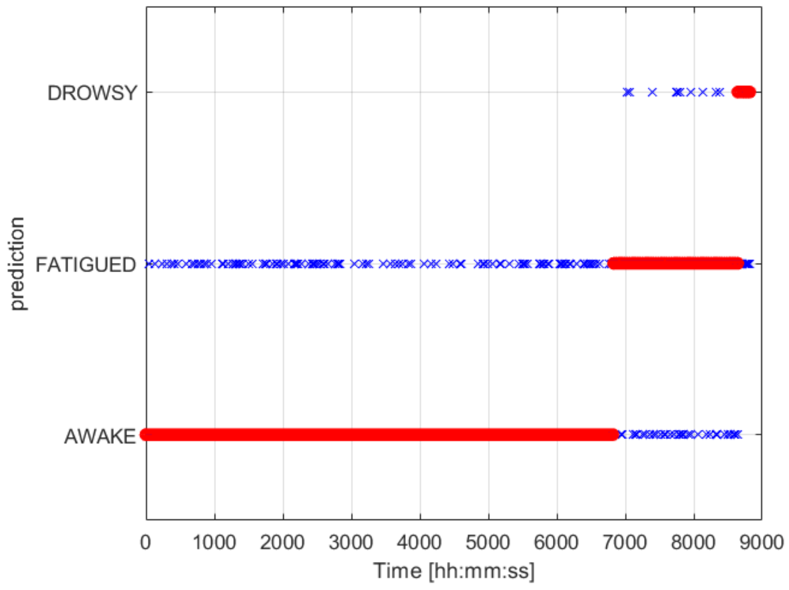


Figure 5.23: test set ('ro': PredictS, 'bx': GRU)

The problem with this method is that the two models don't perform as well as in the Danisi test set, when testing it on additional data coming from real driving simulation, which are not characterized by a balanced distribution of the 3 labels, as in this case. For this reason I prefer the method 1 for training the GRU model, which doesn't perform perfectly on the test set took from the Danisi 23-24 experimentation dataset, but it will allow to increase variability in the training set.

Chapter 6

LSTM model

The LSTM is a type of recurrent neural network architecture designed to model sequential data.
These models introduce memory cells that can retain information over extended sequences, which are controlled by three gates: the input gate, forget gate, and output gate, which regulate the flow of information.

The input gate determines how much of the new input should be stored in the memory cell by processing the current input and the previous hidden state. The forget gate decides which information from the memory cell should be discarded, based on the current input and the previous hidden state. The output gate controls how much information from the memory cell is passed to the hidden state. By using these gates, LSTM networks can selectively maintain, update, and retrieve information over long time periods, making them highly effective for tasks that require long-term dependencies, such as language translation, sentiment analysis, and speech recognition.

The model presents the following sections:

- Input gate: Sequences of shape (300 timesteps, 2 features).
- LSTM layer: Processes the input sequences and has 64 memory units.
- Output Dense Layer: Produces a 3-element vector where each element corresponds to the probability of one of the 3 classes.

Layer (type)	**Output Shape**	**Param #**
lstm_30 (LSTM)	(None, 64)	17,152
dense_30 (Dense)	(None, 3)	195

Table 6.1: type of layers, output shape and number of parameters

As we can see in LSTM layer there're around 4 thousand more parameters with respect to GRU one, which have consequences in the computational complexity of the algorithm running in a physical devices, correlated also with an increased memory occupation and batter consumption [54].

Total params:	17,347
Trainable params:	17,347
Non-trainable params:	0

Table 6.2: trainable and non-trainable parameters

6.1 Method 2: window size of 300 and row input data

I decided to not start from applying the cross-validation method (method 1) for saving time, since it requires multiple iterations, so I exploited the splitting one, or method 2, to train the LSTM model.
I decided to exploit a structure with 64 memory units only and to apply the train settings to the GRU model, in such a way to compare the performances. The only difference was in the patience index in the *early_stopping callback*, set to 5 because I experienced explosion of the loss function after that point.

```
Epoch 1/20 1934/1934 [==============================] - 340s
174ms/step - loss: 0.5705 - accuracy: 0.7729 - val_loss: 0.5207 - val_accuracy: 0.7756 -
lr: 0.0100
Epoch 2/20 1934/1934 [==============================] - 234s
121ms/step - loss: 0.4408 - accuracy: 0.8124 - val_loss: 0.3655 - val_accuracy: 0.8463 -
lr: 0.0090
Epoch 3/20 1934/1934 [==============================] - 417s
216ms/step - loss: 0.3405 - accuracy: 0.8518 - val_loss: 0.3190 - val_accuracy: 0.8622 -
lr: 0.0082
Epoch 4/20 1934/1934 [==============================] - 553s
286ms/step - loss: 0.3344 - accuracy: 0.8562 - val_loss: 0.3353 - val_accuracy: 0.8535 -
lr: 0.0074
Epoch 5/20 1934/1934 [==============================] - 516s
267ms/step - loss: 0.2944 - accuracy: 0.8727 - val_loss: 0.2857 - val_accuracy: 0.8792 -
lr: 0.0067
Epoch 6/20 1934/1934 [==============================] - 274s
142ms/step - loss: 0.2625 - accuracy: 0.8889 - val_loss: 0.2536 - val_accuracy: 0.8898 -
lr: 0.0061
Epoch 7/20 1934/1934 [==============================] - 319s
165ms/step - loss: 0.3421 - accuracy: 0.8566 - val_loss: 0.3259 - val_accuracy: 0.8577 -
lr: 0.0055
Epoch 8/20 1934/1934 [==============================] - 294s
152ms/step - loss: 0.2854 - accuracy: 0.8792 - val_loss: 0.2858 - val_accuracy: 0.8815 -
lr: 0.0050
Epoch 9/20 1934/1934 [==============================] - 288s
149ms/step - loss: 0.3306 - accuracy: 0.8623 - val_loss: 0.4879 - val_accuracy: 0.7799 -
lr: 0.0045
Epoch 10/20 1934/1934 [==============================] - 301s
156ms/step - loss: 0.3698 - accuracy: 0.8335 - val_loss: 0.3474 - val_accuracy: 0.8443 -
lr: 0.0041
Epoch 11/20 1934/1934 [==============================] - 284s
147ms/step - loss: 0.2887 - accuracy: 0.8754 - val_loss: 0.2605 - val_accuracy: 0.8947 -
```

lr: 0.0037

Above I reported the steps of the training through each epoch. In this trial I filled the sliding windows of dimension (WINDOW_SIZE_LONG, 2) with row data (HR and HRV), sampled each second, the scrolled by 1 sample.

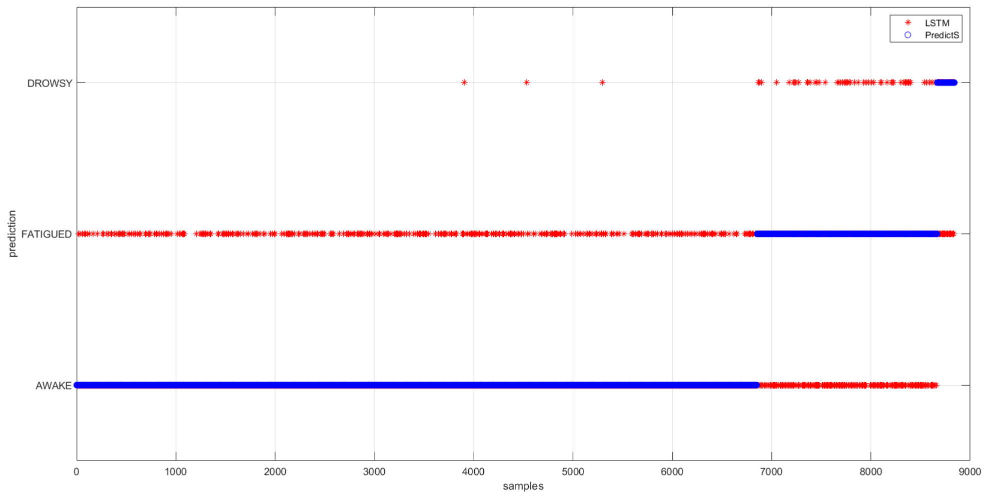


Figure 6.1: test set

Test set (binary): Accuracy: 0.886, Precision: 0.7446, Sensitivity: 0.749, Specificity: 0.9256

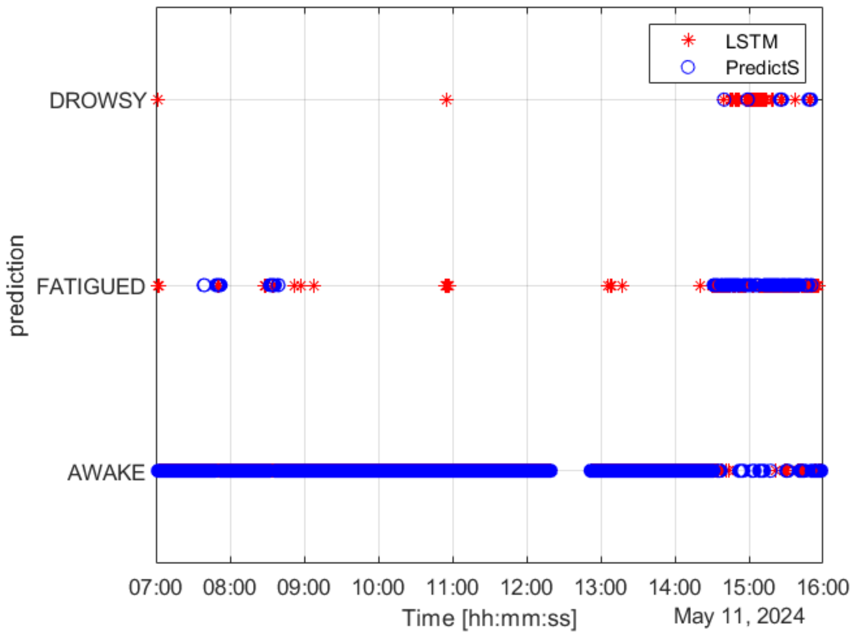


Figure 6.2: MV20240511

Accuracy: 0.93, Precision: 0.742, Sensitivity: 0.63, Specificity: 0.97

As shown from the plots displayed above, since in the first training of the LSTM with row data, in windows of 300 samples, I haven't found any improvements, with respect to the GRU one, I decided to not exploit further its functioning, applying other filtering or windowing methods.

6.2 Techniques to handle unbalancing of data

There are methods to correct the unbalancing of data between levels, to be adapted to the type of problem addressing to.
For example, the data-level approaches include the oversampling of the minority class, the under-sampling of the minority class and the data augmentation.
Between the three I decided to discharge the oversampling, because It consisted of generating synthetic samples for minority classes, by interpolating between existing examples: this approach could fit with the parameter HR, which is almost stable between adiacent intervals (this could be also useful to fill the lack of data I explained when talking about data analysis). The issue is applying this method on the HRV parameter, which can vary not linearly between consecutive samples.
Data augmentation is intended as applying transformations (e.g., jittering, time-shifting, noise injection) to the minority class data to generate more varied samples. About this point I will use this approach, adding huge jitters to the prediction of levels of drowsiness, to test if this can affect the training of the model, in *chapter 7*.
Finally, I decided to exploit the under-sampling the minority class technique, which consists on removing random samples from the over-represented class to balance the dataset. This can help avoid overfitting to the majority class but risks losing valuable information from the majority class.

I started by reducing the total number of samples labeled as AWAKE to its 70% randomly and starting a training from sliding windows of size 300, filled by row data, in order to compare the obtained performances with the ones obtained with the umbalanced dataset.

```
Epoch 1/20 1484/1484 [==============================] - 342s
227ms/step - loss: 0.4161 - accuracy: 0.8226 - val_loss: 0.2424 - val_accuracy: 0.9012 -
lr: 0.0100
Epoch 2/20 1484/1484 [==============================] - 398s
268ms/step - loss: 0.2088 - accuracy: 0.9152 - val_loss: 0.2075 - val_accuracy: 0.9177 -
lr: 0.0090
Epoch 3/20 1484/1484 [==============================] - 397s
268ms/step - loss: 0.1624 - accuracy: 0.9342 - val_loss: 0.1432 - val_accuracy: 0.9466 -
lr: 0.0082
Epoch 4/20 1484/1484 [==============================] - 335s
226ms/step - loss: 0.1285 - accuracy: 0.9483 - val_loss: 0.1264 - val_accuracy: 0.9481 -
lr: 0.0074
Epoch 5/20 1484/1484 [==============================] - 195s
131ms/step - loss: 0.1173 - accuracy: 0.9527 - val_loss: 0.1060 - val_accuracy: 0.9597 -
lr: 0.0067
Epoch 6/20 1484/1484 [==============================] - 227s
153ms/step - loss: 0.1058 - accuracy: 0.9576 - val_loss: 0.1298 - val_accuracy: 0.9436 -
```

```
lr: 0.0061
Epoch 7/20 1484/1484 [==============================] - 219s
148ms/step - loss: 0.1031 - accuracy: 0.9594 - val_loss: 0.1012 - val_accuracy: 0.9628 -
lr: 0.0055
Epoch 8/20 1484/1484 [==============================] - 214s
144ms/step - loss: 0.2991 - accuracy: 0.8791 - val_loss: 0.3172 - val_accuracy: 0.8669 -
lr: 0.0050
Epoch 9/20 1484/1484 [==============================] - 204s
138ms/step - loss: 0.3549 - accuracy: 0.8535 - val_loss: 0.3382 - val_accuracy: 0.8628 -
lr: 0.0045
Epoch 10/20 1484/1484 [==============================] - 222s
150ms/step - loss: 0.2622 - accuracy: 0.8911 - val_loss: 0.2069 - val_accuracy: 0.9175 -
lr: 0.0041
Epoch 11/20 1484/1484 [==============================] - 205s
138ms/step - loss: 0.1748 - accuracy: 0.9281 - val_loss: 0.1470 - val_accuracy: 0.9432 -
lr: 0.0037
Epoch 12/20 1484/1484 [==============================] - 202s
136ms/step - loss: 0.1927 - accuracy: 0.9236 - val_loss: 0.2782 - val_accuracy: 0.8872 -
lr: 0.0033
Epoch 13/20 1484/1484 [==============================] - 194s
131ms/step - loss: 0.2040 - accuracy: 0.9175 - val_loss: 0.1584 - val_accuracy: 0.9343 -
lr: 0.0030
```

The training stopped at the 13th epoch because of an increasing of the validation loss after the 7th epoch. The best model was found at epoch 7th with the following metrics on the test set: accuracy: 0.957, precision: 0.94, sensitivity 0.907, specificity 0.977.

Then I checked the model's performances on the additional registration, by real driving, as follows.

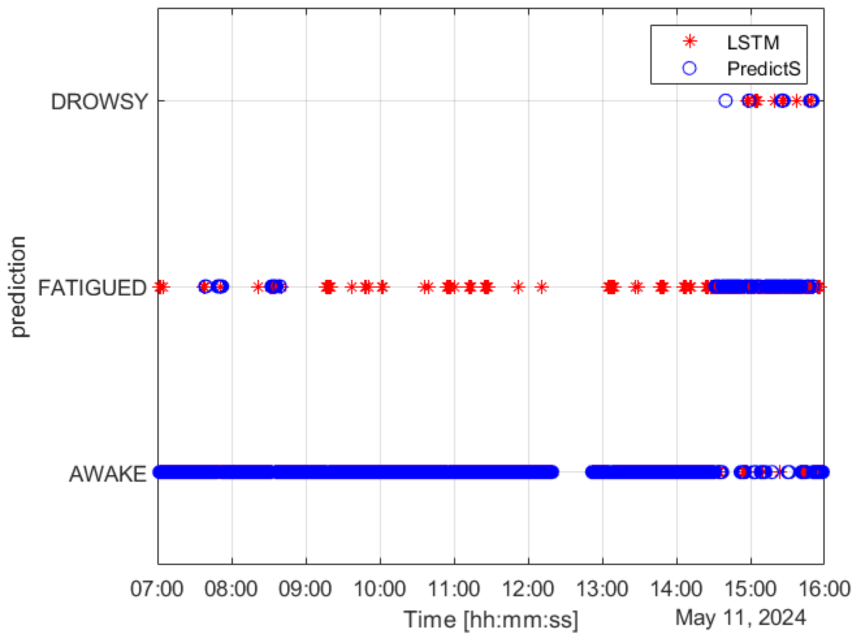


Figure 6.3: MV20240511

The metrics were: accuracy: 0.877, precision: 0.536, sensitivity 0.76, specificity 0.896.

It seems there are no improvements with respect to the original dataset. For this reason I decreased more the total number of samples in AWAKE state, reducing the set size to the sum of samples in ATTENTION and DROWSY state, such that the overall dataset is balanced between drowsy and not-drowsy.
The obtained trained model, used to predict on the real driving dataset, presents the following metrics: accuracy: 0.72, precision: 0.25, sensitivity 0.52, specificity 0.75, which are definitely lower with respect to the original one.
I can conclude that this method is useless to improve the learning of the model in presence of this particular unbalanced dataset.

Chapter 7

Introduction of a random jitter

The next step of this research, after deriving the performances of the GRU model in replying the behaviour of the software PredictS, will be trying to alterate the NN training phase by introducing a random jitter, when reaching the FATIGUED and DROWSY levels.
This method is useful in the sense of reducing the quantity of data, related to the higher levels.
Indeed, between the consequences of imbalanced datasets, there is the possibility for the model to not generalize well on unseen data, mainly if they refers to minority classes, resulting in an high variance and overfitting on the majority classes.
Additionally, the model takes more time to converge, because minority classes are not well represented by training data and metrics such as accuracy can be misleading, because they doesn't take into account the unbalancing.
In fact in my reasoning I will completely ignore the accuracy metrics, in favour of precision, recall, F1-score.

7.1 Delay in predicting FATIGUED and DROWSY

I decided to introduce a variable delay in predicting the FATIGUED and DROWSY levels, which is proportional to the length of the time permanence on that level. This can be done, without deteriorating too much the predictions of the sw PredictS because in principle the drowsy state is predicted with a 5 minutes advance, so that I could use that bounding a margin in the delay.

In particular, I analyzed every file of the Danisi 23 24 experimentation, and when I reached the FATIGUED and DROWSY level from a lower one, I forced the output to remain in the previous level for a number of seconds which varied randomly from the 50% of the time permanence of the prediction on that level, to the 99% in the case of the FATIGUED state, and from the 70% to the 99% in the case of the DROWSY state, controlling that this delay didn't overcome the 300 seconds (5 minutes): in that situation it saturates.
I decided to apply an higher jitter to the DROWSY level with respect to the FATIGUED one, because it has usually short duration, and reducing by the 70% till 99% introduces a delay lower than 5 minutes. Furthermore, it's the most critical level to be tracked and we want to prevent the correct classification of that level.

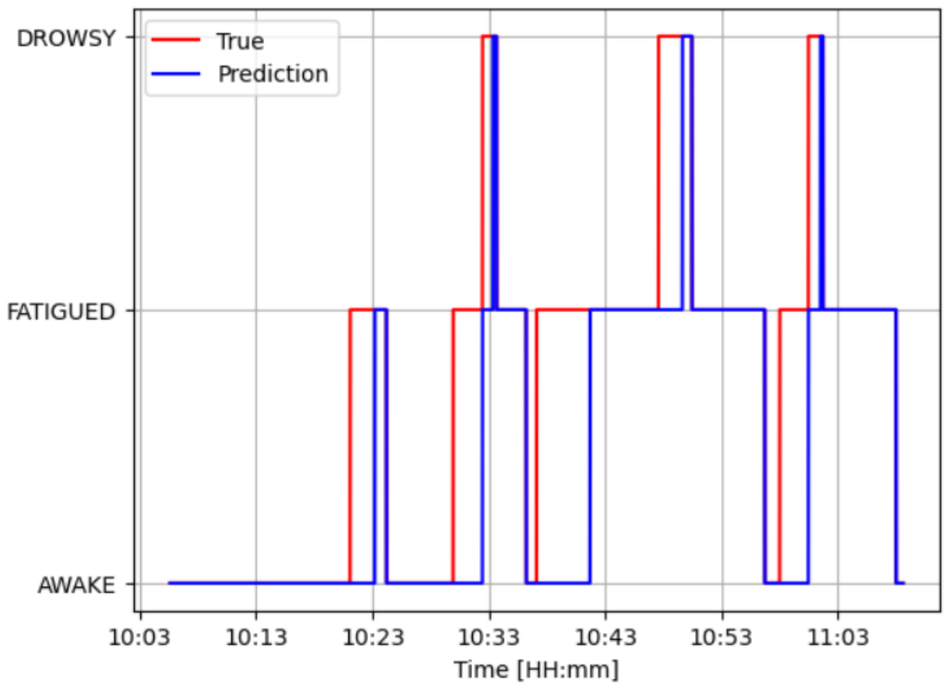


Figure 7.1: subject 2

7.1.1 Method B

Utilizing sliding windows of 300 samples, scrolling of 1 sample every second, filled by row data and training the model with the cross-validation method used before (19-3-2 datasets for training, validation and testing respectively) I obtained the results shown in the table below.

fold	validation sequence	test acc	test prec	test sens	test spec	reach DROWSY
1	[3 4 10]	0.91	0.41	0.107	0.98	No
2	[2 6 8]	0.9	0.36	0.15	0.975	No
3	[0 5 12]	0.88	0.35	0.4	0.93	No
4	[3 6 8]	0.89	0.27	0.147	0.96	1 anticipated
5	[5 8 12]	0.92	0.82	0.167	0.99	No
7	[0 7 11]	0.9	0.2	0.04	0.98	No
8	[0 4 11]	0.87	0.25	0.26	0.92	No
9	[3 9 1]	0.9	0.39	0.21	0.96	No
10	[4 9 11]	0.88	0.31	0.30	0.93	No
11	[0 1 12]	0.91	0.54	0.21	0.98	No
12	[2 7 11]	0.87	0.28	0.28	0.93	No
13	[13 8 10]	0.89	0.35	0.26	0.95	No
14	[6 1 12]	0.92	0.59	0.41	0.97	No
15	[2 3 8]	0.89	0.34	0.31	0.94	No
16	[5 6 1]	0.89	0.35	0.24	0.95	No
17	[4 8 10]	0.92	0.90	0.15	0.99	Neither FATIGUED
18	[0 13 10]	0.89	0.41	0.64	0.91	No
19	[7 13 10]		-	-		Remain in AWAKE
20	[4 13 11]	0.89	0.31	0.17	0.96	No
21	[5 1 8]	0.89	0.34	0.28	0.94	No
22	[3 9 11]	0.9	0.34	0.13	0.7	No
23	[0 2 10]	0.9	0.07	0.01	0.98	Neither FATIGUED
24	[2 1 10]	0.91	0.47	0.33	0.96	No
25	[5 9 10]	0.91	0.49	0.45	0.95	No
26	[2 6 12]	0.9	0.38	0.29	0.95	No
27	[5 7 11]	0.88	0.39	0.76	0.89	No
28	[3 7 11]	0.9	0.45	0.33	0.96	No
29	[3 4 8]	0.9	0.45	0.33	0.96	Neither FATIGUED
30	[4 5 12]	0.88	0.37	0.58	0.91	No
31	[13 10 11]	0.91	0.48	0.34	0.96	No
32	[5 8 10]	0.9	0.19	0.05	0.97	Neither FATIGUED
33	[4 13 8]	0.9	0.36	0.17	0.97	Neither FATIGUED
34	[2 1 11]	0.9	0.35	0.13	0.97	Neither FATIGUED
35	[5 7 12]	0.86	0.3	0.46	0.9	Neither FATIGUED
36	[5 7 10]	0.87	0.36	0.68	0.89	1 but late
37	[0 9 12]	0.87	0.06	0.035	0.95	No
38	[0 7 8]	0.93	0.78	0.31	0.99	No
39	[2 1 8]	0.92	0.53	0.62	0.94	No

Table 7.1: Method B when introducing a random delay in transitions from lower to higher levels of drowsiness: example of the first 39 folds

7.1.2 Method A

Introducing the random jitter in transitioning from AWAKE to FATIGUED state and from FATIGUED to DROWSY, and applying the method 2 for splitting the sliding windows, used to feed the GRU model, I obtained the following results.
If we reason on the precision and sensitivity metrics, which are the ones relevant when dealing with unbalanced datasets, we can see a relevant drop of the performances, when injecting a random delay in notifying the medium and high levels of drowsiness, even if I've not added a delay when going from higher to lower yet.
This lets me think that reasonably when I'll also inject a random jitter in transitions from higher to lower levels of drowsiness, the performances will deteriorate more.

Details of the training:

```
Epoch 1/20 1934/1934 [==============================] - 267s
136ms/step - loss: 0.2417 - accuracy: 0.9094 - val_loss: 0.1879 - val_accuracy: 0.9266 -
lr: 0.0100
Epoch 2/20 1934/1934 [==============================] - 313s
162ms/step - loss: 0.1510 - accuracy: 0.9436 - val_loss: 0.1332 - val_accuracy: 0.9467
- lr: 0.0090
Epoch 3/20 1934/1934 [==============================] - 322s
166ms/step - loss: 0.1877 - accuracy: 0.9289 - val_loss: 0.1954 - val_accuracy: 0.9285 -
lr: 0.0082
Epoch 4/20 1934/1934 [==============================] - 341s
176ms/step - loss: 0.1773 - accuracy: 0.9326 - val_loss: 0.1479 - val_accuracy: 0.9473 -
lr: 0.0074
Epoch 5/20 1934/1934 [==============================] - 214s
111ms/step - loss: 0.1813 - accuracy: 0.9322 - val_loss: 0.2136 - val_accuracy: 0.9162 -
lr: 0.0067
Epoch 6/20 1934/1934 [==============================] - 202s
104ms/step - loss: 0.2180 - accuracy: 0.9182 - val_loss: 0.3277 - val_accuracy: 0.8799 -
lr: 0.0061
Epoch 7/20 1934/1934 [==============================] - 204s
106ms/step - loss: 0.2795 - accuracy: 0.8930 - val_loss: 0.2702 - val_accuracy: 0.8960 -
lr: 0.0055
Epoch 8/20 1934/1934 [==============================] - 245s
127ms/step - loss: 0.2563 - accuracy: 0.9013 - val_loss: 0.2517 - val_accuracy: 0.9028 -
lr: 0.0050
```

Testing the model on the test set from Danisi 23-24 (merging the 10% of the whole windows referred to label 1,2,3) I obtained:
accuracy: 0.946
precision: 0.794
sensitivity 0.72
specificity 0.975.
The accuracy values are similar for the training, validation and test sets, which means that the model is not supposed to be affected by the overfitting problem.
As explained before, this is a better method, with respect to cross-validation, to try to correct the unbalancing of data between levels, for this reason, since each level is better

represented, the performance of the final model increases.

Adding the jitter helps to degrade the precision and sensitivity, which are the indexes relevant when dealing with classification problems, affected by umbalancing of data, but this indices are strictly dependent on how casually the three sets are split: in fact re-running the splitting, the sets changes, with also the metrics.

If we reason on the precision and sensitivity metrics, which are the ones relevant when dealing with unbalanced datasets, we can see a relevant drop of the performances, when injecting a random delay in notifying the medium and high levels of drowsiness, besides adding a delay when going from higher to lower yet.
This lets me think that reasonably when I'll also inject a random jitter in transitions from higher to lower levels of drowsiness, the performances will deteriorate more.

7.2 Delay in predicting higher levels of drowsiness and in restoring the lower ones

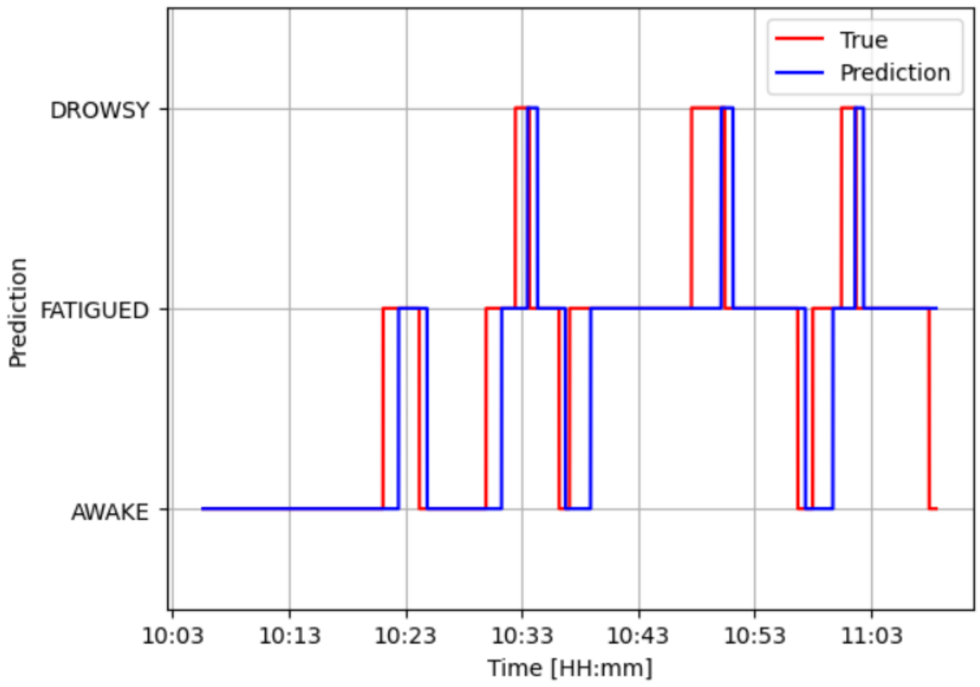


Figure 7.2: subject 2

I tried to get the model learning worse by also adding a random delay between 30 and 60 seconds, in the transition between DROWSY-FATIGUED and FATIGUED-AWAKE, besides the one already added when transitioning from lower levels of drowsiness to higher.

Details of the training:

Epoch 1/20 1934/1934 [==============================] - 211s 108ms/step - loss: 0.4181 - accuracy: 0.8236 - val_loss: 0.4273 - val_accuracy: 0.8144 -

```
lr: 0.0100
Epoch 2/20 1934/1934 [==============================] - 201s
104ms/step - loss: 0.4172 - accuracy: 0.8181 - val_loss: 0.4056 - val_accuracy: 0.8226 -
lr: 0.0090
Epoch 3/20 1934/1934 [==============================] - 192s
99ms/step - loss: 0.3904 - accuracy: 0.8281 - val_loss: 0.3904 - val_accuracy: 0.8271 - lr:
0.0082
Epoch 4/20 1934/1934 [==============================] - 196s
101ms/step - loss: 0.3602 - accuracy: 0.8401 - val_loss: 0.3454 - val_accuracy: 0.8465 -
lr: 0.0074
Epoch 5/20 1934/1934 [==============================] - 216s
112ms/step - loss: 0.3286 - accuracy: 0.8560 - val_loss: 0.3142 - val_accuracy: 0.8662 -
lr: 0.0067
Epoch 6/20 1934/1934 [==============================] - 208s
108ms/step - loss: 0.3830 - accuracy: 0.8382 - val_loss: 0.4683 - val_accuracy: 0.7975 -
lr: 0.0061
Epoch 7/20 1934/1934 [==============================] - 221s
114ms/step - loss: 0.4047 - accuracy: 0.8300 - val_loss: 0.3833 - val_accuracy: 0.8393 -
lr: 0.0055
Epoch 8/20 1934/1934 [==============================] - 173s
89ms/step - loss: 0.3973 - accuracy: 0.8330 - val_loss: 0.3757 - val_accuracy: 0.8397 - lr:
0.0050
Epoch 9/20 1934/1934 [==============================] - 207s
107ms/step - loss: 0.3476 - accuracy: 0.8507 - val_loss: 0.3193 - val_accuracy: 0.8622 -
lr: 0.0045
Epoch 10/20 1934/1934 [==============================] - 196s
101ms/step - loss: 0.3055 - accuracy: 0.8716 - val_loss: 0.3617 - val_accuracy: 0.8485 -
lr: 0.0041
Epoch 11/20 1934/1934 [==============================] - 189s
97ms/step - loss: 0.3596 - accuracy: 0.8508 - val_loss: 0.3473 - val_accuracy: 0.8592 - lr:
0.0037
```

Testing the model on the test set from Danisi 23-24 (merging the 10% of the whole windows referred to label 1,2,3) I obtained:
accuracy: 0.863
precision: 0.727
sensitivity: 0.46
specificity 0.95

For what concerns the training of this model with non-experimental datasets, we obtain for MV20240511 the metrics:
precision: 0.65
sensitivity 0.246

7.3 Delay in predicting levels of drowsiness and revise the data related to the FATIGUED level

At this point I made the last attempt in trying to confuse the NN in learning from the sliding windows, by adding a random interval between 30 and 60 seconds, every random number of seconds between 180 and 240 (every 2-3 minutes) in which the Predicts output remains in the FATIGUED state.

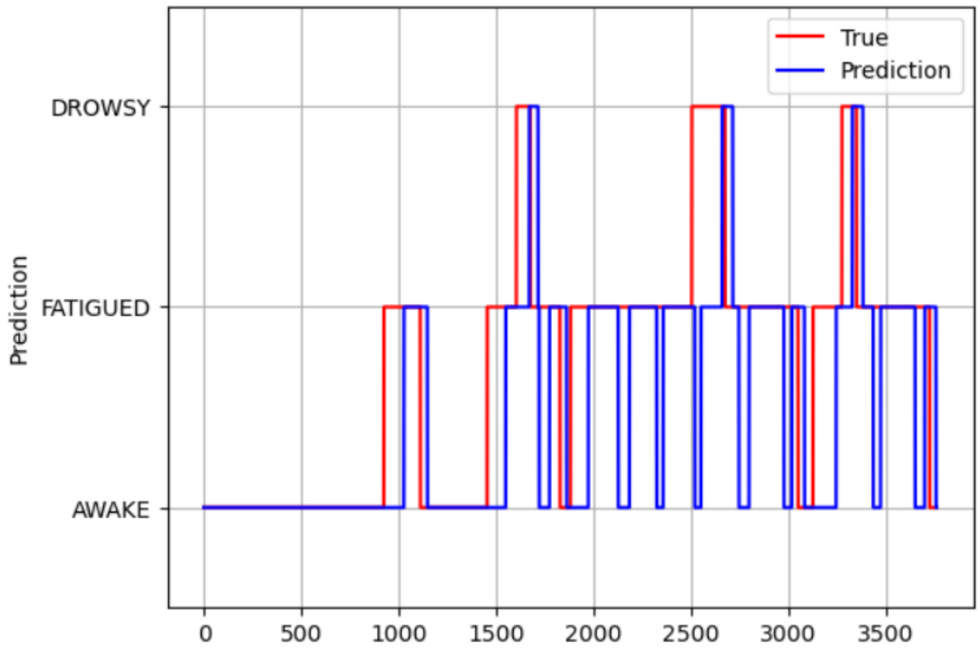


Figure 7.3: subject 2 (x axis: number of samples)

Details of the training:

Epoch 1/20 1934/1934 [==============================] - 184s 94ms/step - loss: 0.3647 - accuracy: 0.8518 - val_loss: 0.4348 - val_accuracy: 0.8344 - lr: 0.0100
Epoch 2/20 1934/1934 [==============================] - 182s 94ms/step - loss: 0.4225 - accuracy: 0.8333 - val_loss: 0.4053 - val_accuracy: 0.8354 - lr: 0.0090
Epoch 3/20 1934/1934 [==============================] - 183s 94ms/step - loss: 0.4121 - accuracy: 0.8347 - val_loss: 0.4053 - val_accuracy: 0.8342 - lr: 0.0082
Epoch 4/20 1934/1934 [==============================] - 183s 95ms/step - loss: 0.4075 - accuracy: 0.8345 - val_loss: 0.3986 - val_accuracy: 0.8351 - lr: 0.0074
Epoch 5/20 1934/1934 [==============================] - 181s 94ms/step - loss: 0.4053 - accuracy: 0.8356 - val_loss: 0.3975 - val_accuracy: 0.8362 - lr: 0.0067
Epoch 6/20 1934/1934 [==============================] - 182s 94ms/step - loss: 0.4044 - accuracy: 0.8355 - val_loss: 0.4015 - val_accuracy: 0.8354 - lr:

```
0.0061
Epoch 7/20 1934/1934 [==============================] - 182s
94ms/step - loss: 0.4024 - accuracy: 0.8357 - val_loss: 0.3980 - val_accuracy: 0.8370 - lr:
0.0055
Epoch 8/20 1934/1934 [==============================] - 181s
94ms/step - loss: 0.4020 - accuracy: 0.8351 - val_loss: 0.4016 - val_accuracy: 0.8356 - lr:
0.0050
Epoch 9/20 1934/1934 [==============================] - 184s
95ms/step - loss: 0.4007 - accuracy: 0.8357 - val_loss: 0.4017 - val_accuracy: 0.8357 - lr:
0.0045
Epoch 10/20 1934/1934 [==============================] - 184s
95ms/step - loss: 0.4009 - accuracy: 0.8353 - val_loss: 0.3965 - val_accuracy: 0.8347 - lr:
0.0041
Epoch 11/20 1934/1934 [==============================] - 372s
192ms/step - loss: 0.4001 - accuracy: 0.8362 - val_loss: 0.3950 - val_accuracy: 0.8344 -
lr: 0.0037
Epoch 12/20 1934/1934 [==============================] - 217s
112ms/step - loss: 0.3996 - accuracy: 0.8354 - val_loss: 0.3989 - val_accuracy: 0.8363 -
lr: 0.0033
Epoch 13/20 1934/1934 [==============================] - 219s
113ms/step - loss: 0.3986 - accuracy: 0.8359 - val_loss: 0.3993 - val_accuracy: 0.8349 -
lr: 0.0030
Epoch 14/20 1934/1934 [==============================] - 267s
138ms/step - loss: 0.3983 - accuracy: 0.8352 - val_loss: 0.3941 - val_accuracy: 0.8353 -
lr: 0.0027
Epoch 15/20 1934/1934 [==============================] - 185s
95ms/step - loss: 0.3979 - accuracy: 0.8354 - val_loss: 0.3952 - val_accuracy: 0.8374 - lr:
0.0025
Epoch 16/20 1934/1934 [==============================] - 188s
97ms/step - loss: 0.3975 - accuracy: 0.8357 - val_loss: 0.3946 - val_accuracy: 0.8355 - lr:
0.0022
Epoch 17/20 1934/1934 [==============================] - 193s
100ms/step - loss: 0.3978 - accuracy: 0.8353 - val_loss: 0.3952 - val_accuracy: 0.8363 -
lr: 0.0020
Epoch 18/20 1934/1934 [==============================] - 189s
98ms/step - loss: 0.3972 - accuracy: 0.8357 - val_loss: 0.3967 - val_accuracy: 0.8356 - lr:
0.0018
Epoch 19/20 1934/1934 [==============================] - 190s
98ms/step - loss: 0.3969 - accuracy: 0.8358 - val_loss: 0.3972 - val_accuracy: 0.8360 - lr:
0.0017
Epoch 20/20 1934/1934 [==============================] - 190s
98ms/step - loss: 0.3966 - accuracy: 0.8356 - val_loss: 0.3936 - val_accuracy: 0.8358 -
lr: 0.0015
```

Testing the model on the test set from Danisi 23-24 (merging the 10% of the whole windows referred to label 1,2,3) I obtained:
accuracy: 0.835
precision: 0.5
sensitivity: 0.1
specificity 0.97

For what concerns the training of this model with non-experimental datasets, we obtain for MV20240511 the metrics:
precision: 0.0426
sensitivity 0.0257

We can derive that this method is effective in obtain low sensitivity performances, when trying to replicate the same behaviour of the sw.
In particular, the examples displayed previously are derived by running the training of the RNN with different random delays, finding the combinations, which deteriorates the performances the most. Indeed, there are some cases when, even if inserting a jitter It doesn't ruin the learning of the model.

At this point it will be time for the company to reason about if It's convenient to modify the PredictS outputs and relative performances, in order to obtain on the other hand a safer environment, when talking about reverse engineering of the algorithm.

Chapter 8

Implementation of the model

Once obtained the model, which implements some of the functionalities of the PredictS software, the next step is to implement it on an Android application, in order to analyze how the computational complexity of the NN can be adapted to a real scenario (in terms of smartphone battery consuption, memory usage, ...).

8.1 Set-up

I started from the Android Studio app already developed for implementing PredictS and added some adjustments. In order to add TensorFlow Lite dependencies in the Android project using Android Studio, I modified the **build.gradle** file for the app module, adding the following lines of code in the block **dependencies**

```
implementation("org.tensorflow:tensorflow-lite:2.10.0")
implementation("org.tensorflow:tensorflow-lite-support:0.3.1")
implementation("org.tensorflow:tensorflow-lite-select-tf-ops:2.10.0")
```

The first one is the main TensorFlow Lite library that provides the core functionalities for running ML models on mobile and edge devices. It includes the necessary runtime to execute TensorFlow Lite models.
The TensorFlow Lite version 2.10.0 is fully compatible with the TensorFlow version installed in my Anaconda environment, which I use for training the model with the Python kernel. Ensuring that both versions match guarantees smooth integration and optimal performance when deploying the trained model on Android devices.
By aligning the versions of TensorFlow Lite in the Android project and TensorFlow in the Anaconda environment, we can ensure that the training and inference processes are consistent and reliable.
The third library allows the use of select TensorFlow operations that are not included in the standard TensorFlow Lite runtime. It is useful when your model contains operations that are not natively supported by TensorFlow Lite but are available in the full TensorFlow framework.

8.1.1 TensorFlow Lite model

The **onCreate** method of the **MainActivity** class is where I initialized the Prediction class. This method is the entry point for the activity: it's called when he activity is first

created, and it's where to perform basic application startup logic.
Since the RNN model must be loaded before performing any of the classification functionalities, for this reason, it's important to call the **initialize** method of the **prediction** instance of the **Prediction** class when the activity is created.

```kotlin
class MainActivity : AppCompatActivity(), DevicePairedStateListener,
   DeviceConnectionStateListener, PairingStatusListener {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            val prediction = Prediction()
            prediction.initialize(this)
            ...
            }
    ...
}
```

By calling **initialize(this)** in the **onCreate** method, you ensure that the Prediction object is set up and ready to perform predictions as soon as your activity starts. This includes having the TensorFlow Lite model loaded and having the necessary data read and stored.
This setup is crucial for real-time applications where predictions need to be made continuously as new data comes in. By initializing everything in **onCreate**, I ensure that there are no delays or setup steps required later when predictions are needed.

```kotlin
class Prediction: ValueChangeListenerDeviceConnected,
   ValueChangeListenerServiceStopped {
    ...
    //TF model
    lateinit var tflite: Interpreter
    var isModelLoaded = false

    @Throws(IOException::class)
    private fun loadModelFile(context: Context): MappedByteBuffer {
        val fileDescriptor = context.assets.openFd("model.tflite")
        val inputStream =
   FileInputStream(fileDescriptor.fileDescriptor)
        val fileChannel = inputStream.channel
        val startOffset = fileDescriptor.startOffset
        val declaredLength = fileDescriptor.declaredLength
        return fileChannel.map(FileChannel.MapMode.READ_ONLY,
   startOffset, declaredLength)
    }

    @Throws(IOException::class)
    private fun createInterpreter(context: Context): Interpreter {
        val tfliteModel = loadModelFile(context)
        return Interpreter(tfliteModel)
    }

    fun loadModel(context: Context) {
        GlobalScope.launch(Dispatchers.IO) {
            try {
                tflite = createInterpreter(context)
                // Imposta isModelLoaded su true dopo aver caricato
   correttamente il modello
                Log.d("Prediction", "Model loaded successfully")
                isModelLoaded = true
            } catch (e: IOException) {
                e.printStackTrace()
                Log.e("Prediction", "Failed to load model", e)
            }
        }
    }

    fun initialize(context: Context) {
        loadModel(context)
    }
    ...
}
```

This is the fragment of code in the **Prediction** class involved in the initialization of the TF model. The explanation of each part follows:

- Class-level variables:
  tflite is a variable that will hold the TensorFlow Lite interpreter instance; the attribute *lateinit* means that this variable will be initialized lated before it's used.
  isModelLoaded is a flag to indicate whether the TensorFlow Lite model has been successfully loaded. It's initialized as *false* and is set to *true* once the model is loaded.

- Method to Load the Model File: private fun *loadModelFile(context: Context): MappedByteBuffer*:
  This method loads the TensorFlow Lite model file from the assets folder. It takes a **Context** object as a parameter to access the app's assets.
  It computes the following steps:

  1. *openFd("model.tflite")*: opens a file descriptor for the model.tflite file located in the assets folder.
  2. *FileInputStream(fileDescriptor.fileDescriptor)*: creates an InputStream for the file.
  3. *inputStream.channel*: gets the file channel to read the file.
  4. *fileChannel.map(...)*: maps the file into memory for fast access. This creates a *MappedByteBuffer* which is returned.

- Method to Create the Interpreter: private fun *createInterpreter(context: Context): Interpreter*:
  This method creates an instance of the TensorFlow Lite interpreter using the model file, taking a **Context** object as a parameter to load the model file. It computes the following steps:

  1. *loadModelFile(context)*: it calls the *loadModelFile* method to get the *MappedByteBuffer* of the model.
  2. *Interpreter(tfliteModel)*: it creates a new Interpreter instance using the model buffer.

- Method to Load the Model Asynchronously: fun *loadModel(context: Context)*: It's in charge of loading the model in a background thread to avoid blocking the main UI thread.
  Steps:

  1. *GlobalScope.launch(Dispatchers.IO)*: it launches a coroutine in the IO dispatcher (background thread).
  2. *createInterpreter(context)*: it calls the *createInterpreter* method to create the interpreter.
  3. *Log.d("Prediction", "Model loaded successfully")*: it logs a message indicating the model was loaded successfully.
  4. *isModelLoaded* = true: it sets the *isModelLoaded* flag to true.
  5. *catch (e: IOException)*: it catches any IOException that occurs during loading, logs the error, and prints the stack trace.

- Initialization Method: fun *initialize(context: Context)*: This method serves as a single entry point to initialize the *Prediction* object by loading the model.

8.1.2 Behaviour of the class Prediction: management of input data and predictions

The Prediction class is designed to predict the drowsiness state of the driver, based on real-time and historical health data: HR and HRV variables, sampled at 1Hz frequency from the PPG sensor of the smartwatch device. The TensorFlow Lite model, initialized previously, is implemented to predict if the driver's level of drowsiness or attention. The class implements two interfaces, ValueChangeListenerDeviceConnected and ValueChange-ListenerServiceStopped, which are used for managing device connectivity and service status changes.

```
class Prediction: ValueChangeListenerDeviceConnected,
   ValueChangeListenerServiceStopped {
    ...

    private val inputSize = 300 #input size
    private val numFeatures = 2 # Number of features per input
    private val outputSize = 3 # Number of lables for the output

    #Z-score normalization of input data: mean and std of training set
    private val TrainSetMean= listOf(72.27778918, 43.69162515)
    private val TrainSetStd= listOf(9.6385747,  32.56838213)

    private var index10=0 #counter for 10 samples buffer
    private var index300=0 #counter for 300 samples buffer
    private val buffer10 = Array(10) { DoubleArray(numFeatures) }
    private val input= Array(1){Array(inputSize){
   FloatArray(numFeatures)}}

    private var compute_output=false #a flag to indicate when the
   model can be run
```

```

    fun findIndexOfMaxAbsValue(array: FloatArray): Int {
        return array.indices.maxByOrNull { abs(array[it]) }  ?: -1
    } #finds the index which contains the gratest probability in the output vector (sum of probabilities=1)

    private var first10 = false
    private var first300 = false

    fun initialize(context: Context) {
        loadModel(context)
    }

     private fun predict(realtimeData: REALTIME_DATA, healthData: HEALTH_DATA, reset: Int, userData: USER_DATA): DROWSINESS_STATE {
        val output = Array(1) { FloatArray(outputSize) }
        val realtimeHR = realtimeData.HR
        val realtimeHRV = realtimeData.HRV

        if (realtimeHR > 30) {
            handleBuffering(realtimeHR, realtimeHRV)

            if (computeOutput && isModelLoaded) {
                tflite.run(input, output)
                output[0]?.run {
                    val indexPrediction = findIndexOfMaxAbsValue(this)
                    return when (indexPrediction) {
                        0 -> DROWSINESS_STATE.AWAKE
                        1 -> DROWSINESS_STATE.FATIGUED
                        2 -> DROWSINESS_STATE.DROWSY
                        -1 -> DROWSINESS_STATE.LOWDATAQUALITY
                        else -> DROWSINESS_STATE.CALIBRATE
                    }
                }
            } else {
                return DROWSINESS_STATE.CALIBRATE
            }
        } else {
            return DROWSINESS_STATE.LOWDATAQUALITY
        }
    }

    private fun handleBuffering(hr: Double, hrv: Double) {
        if (index10 < 10 && !first10) {
            buffer10[index10][0] = hr
            buffer10[index10][1] = hrv
            index10++
        }
        if (index10 == 10) {
        #slides the data of 1 place from left to right delating the oldest: sliding window of 10 samples
            if (first10) {
                slideBuffer(buffer10)
        #adds the 10th sample (for all windows after the first to be filled)
                buffer10[9][0] = hr
                buffer10[9][1] = hrv
            }
            first10 = true

            #deploys the sample in the second buffer of size 300
            val col1 = buffer10.map { it[0] }
            val col2 = buffer10.map { it[1] }

            #data normalization
            val col1Normalized = ((col1.average() - trainSetMean[0]) / trainSetStd[0]).toFloat()
            val col2Normalized = ((col2.standardDeviation() - trainSetMean[1]) / trainSetStd[1]).toFloat()

            if (index300 < inputSize && !first300) {
                input[0][index300][0] = col1Normalized
                input[0][index300][1] = col2Normalized
                index300++
            }
            if (index300 == inputSize) {
                if (first300) {
              #slides the data of 1 place from left to right delating the oldest: sliding window of 300 samples
                    slideBuffer(input[0])
                    #adds the (buffer.size)th sample (for all windows after the first to be filled)
                    input[0][inputSize - 1][0] = col1Normalized
                    input[0][inputSize - 1][1] = col2Normalized
                }
                first300 = true
                computeOutput = true
            }
        }
    }

    private fun slideSmallBuffer(buffer: Array<DoubleArray>) {
    #slides the data of 1 place from left to right delating the oldest: sliding window of n samples
        for (i in 1 until buffer.size) {
        for (j in 0 until numFeatures) {
                    buffer[i - 1][j] = buffer[i][j]
            }
        }
    }

    private fun slideInputBuffer(buffer: Array<Array<FloatArray>>) {
        for (i in 1 until inputSize) {
            for (j in 0 until numFeatures) {
                buffer[0][i - 1][j] = buffer[0][i][j]
            }
        }
    }
}
```

The Prediction class handles real-time data buffering, normalization, and prediction using a pre-trained AI model for drowsiness detection.

Methods:

- *initialize*: loads the AI model and initializes the *totData* array reading the data from a file. This array will store HR and HRV values, sampled every second, from a known dataset, in such a way to test the application behaviour with respect to known predictions from the PredictS sw.
- *predict*: main method for making predictions. Handles data buffering and runs inference if the buffers are properly filled and the model is loaded. Returns the predicted drowsiness state. The predict method processes incoming real-time data coming from Garmin smartwatch (realtimeData). If the heart rate (*realtimeHR*) is greater than 30, which is a physiologically reasonable value, it calls *handleBuffering* to manage data buffering. If the *computeOutput* flag is true and the model is loaded, it runs the prediction and returns the drowsiness state based on the highest probability in the model's output. If the condition on the quality of *realtimeHR* is not respected, it returns as predicted output DROWSINESS_STATE.LOWDATAQUALITY and executes the following sample.
- *handleBuffering*: manages the buffering of incoming HR and HRV data. Updates both the 10-sample buffer (buffer10) and the 300-sample buffer (input). At first it updates buffer10 with the latest HR and HRV values. When buffer10 is filled, it computes the average and standard deviation, normalizes these values, and updates the input buffer. Then it sets *computeOutput* to true when the 300-sample buffer is filled, in such a way it can be processed into *predict* function.
- *slideSmallBuffer*: helper function to slide the buffer10 window, discarding the oldest sample and making room for a new sample.
- *slideInputBuffer*: helper function to slide the *input* window (300-sample buffer), discarding the oldest sample and making room for a new sample. This structure ensures that the model always has a recent, normalized window of data to make accurate predictions.

8.2 Testing of the application in terms of prediction

I tested the app implementing the GRU model algorithm method 2 (pre-processed HR and HRV in sliding windows of 10 filling the input size of the RNN model (300,2)) by uploading a file *.txt* of a registration used for testing the NN on Python environment into the *AndroidStudio* folder.
In this way I checked the same behaviour on both application log files and on Python script.

```
#array where the data used for testing the application will be saved
    val totData = Array(1) { Array(5631) { DoubleArray(numFeatures) } }

    private var cont=0 #couter to iterate through totData vector when
   testing the application
```

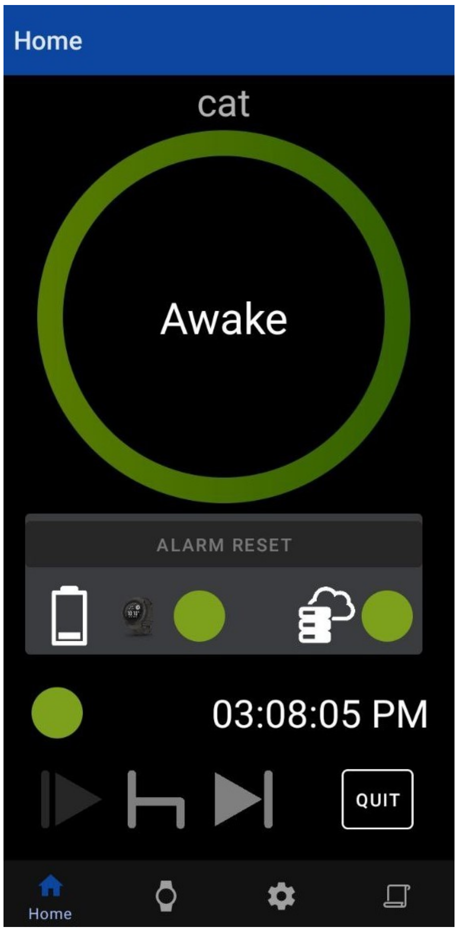
In class Prediction I defined an array (1,5631,2) to save the registration I wanted to test, coming from Danisi 2024, subject 22. Then initialize to 0 the counter which iterates in this array.

```
fun initialize(context: Context) {
        loadModel(context)
        #reads the file used for test the application:
        val text = readTextFileFromAssets(context,"text.txt")
        var l=0
        for (pair in text) {
            val (value1, value2) = pair
            totData[0][l][0] = value1.toDouble()
            totData[0][l][1] = value2.toDouble()
            l++
        }
    }
```

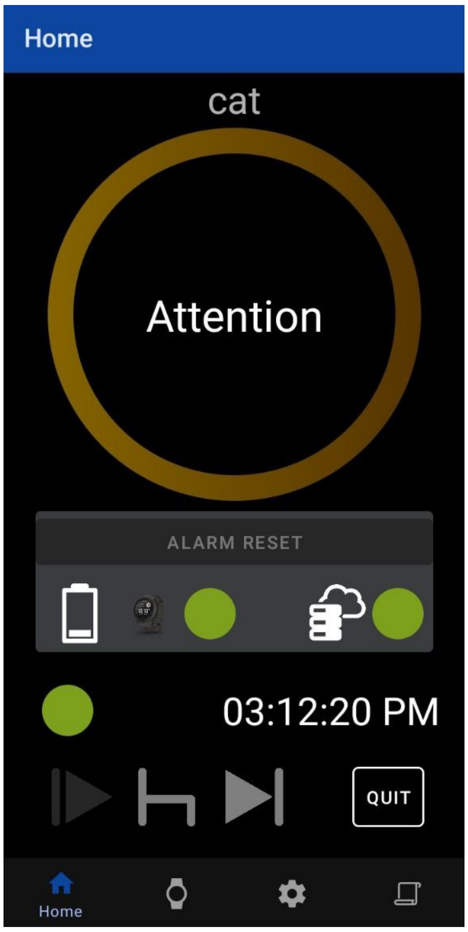
In the function *initialize* add the reading of the file: this function is called at first when starting the *Activity*.

```
val realtimeHR=totData[0][cont][0]
val realtimeHRV=totData[0][cont][1]
```

At each calling of the *predict* function, with a frequency of 1 sample/sec, instead of initializing the variables in the predict function to the real time value of HR and HRV, they are initialized to the count(th) value of *totData* Then the counter *count*, which iterates through the *totData* vector is increased.



(a) user interface in AWAKE



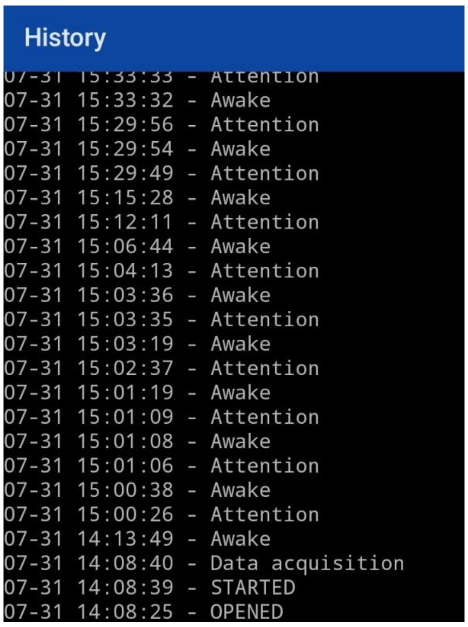
(b) user interface in ATTENTION (FATIGUED)

Figure 8.1: user interface displaying the predicted drowsiness state

Each window of lenght 300 is associated with an output of the model: this means if we want to create a correlation between the starting of the data acquisition and the first prediction (first of the k_{th} examples), we have to sum 309 unlabelled seconds (the ones spennt to fill the buffer in the *calibration* phase), in which the two buffers cooperate in view of obtain the first *(300,2)* vector, to serve as input of the NN.

window	from	to	window	from	to
2797	AWAKE	ATTENTION	4565	ATTENTION	AWAKE
2809	ATTENTION	AWAKE	4567	AWAKE	ATTENTION
2837	AWAKE	ATTENTION	4783	ATTENTION	AWAKE
2839	ATTENTION	AWAKE	4784	AWAKE	ATTENTION
2840	AWAKE	ATTENTION	4786	ATTENTION	AWAKE
2850	ATTENTION	AWAKE	5002	AWAKE	ATTENTION
2928	AWAKE	ATTENTION	5019	ATTENTION	AWAKE
2970	ATTENTION	AWAKE	5051	AWAKE	ATTENTION
2986	AWAKE	ATTENTION	5125	ATTENTION	AWAKE
2987	ATTENTION	AWAKE	5136	AWAKE	ATTENTION
3024	AWAKE	ATTENTION	5138	ATTENTION	AWAKE
3175	ATTENTION	AWAKE	5140	AWAKE	ATTENTION
3502	AWAKE	ATTENTION	5142	ATTENTION	AWAKE
3699	ATTENTION	AWAKE	5149	AWAKE	ATTENTION
4560	AWAKE	ATTENTION	5151	ATTENTION	AWAKE

Table 8.1: label transitions at the k_{th} window


History
```
07-31 15:33:33 - Attention
07-31 15:33:32 - Awake
07-31 15:29:56 - Attention
07-31 15:29:54 - Awake
07-31 15:29:49 - Attention
07-31 15:15:28 - Awake
07-31 15:12:11 - Attention
07-31 15:06:44 - Awake
07-31 15:04:13 - Attention
07-31 15:03:36 - Awake
07-31 15:03:35 - Attention
07-31 15:03:19 - Awake
07-31 15:02:37 - Attention
07-31 15:01:19 - Awake
07-31 15:01:09 - Attention
07-31 15:01:08 - Awake
07-31 15:01:06 - Attention
07-31 15:00:38 - Awake
07-31 15:00:26 - Attention
07-31 14:13:49 - Awake
07-31 14:08:40 - Data acquisition
07-31 14:08:39 - STARTED
07-31 14:08:25 - OPENED
```

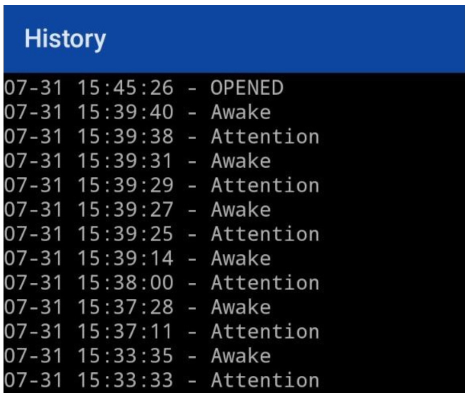

History
```
07-31 15:45:26 - OPENED
07-31 15:39:40 - Awake
07-31 15:39:38 - Attention
07-31 15:39:31 - Awake
07-31 15:39:29 - Attention
07-31 15:39:27 - Awake
07-31 15:39:25 - Attention
07-31 15:39:14 - Awake
07-31 15:38:00 - Attention
07-31 15:37:28 - Awake
07-31 15:37:11 - Attention
07-31 15:33:35 - Awake
07-31 15:33:33 - Attention
```

Figure 8.2: history of label transitions

Comparing the history of the level of drowsiness transitions computed "real-time" on the app, with the one obtained running the model on Visual Studio, with a Python script, the behaviour matches.

8.3 Memory usage and battery consumption

8.3.1 Memory usage

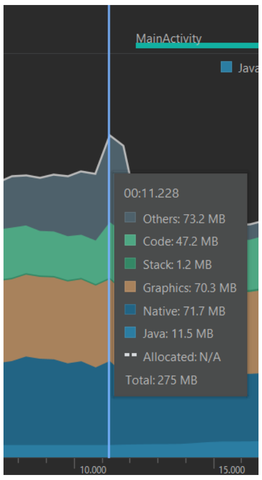
Memory Profiler and *CPU Profiler* are tools provided by Android Studio, in order to monitor and study the developing app behaviour.
In particular the *Memory Profiler* focuses on monitoring and optimizing memory usage to prevent memory leaks and reduce memory consumption and the *CPU Profiler* focuses on analyzing and optimizing CPU usage to improve app performance and responsiveness. The *CPU Profiler* tool allows the user to monitor the developed app's CPU consumption and thread behavior live during app interaction or review recorded method, function, and system traces for detailed analysis.
Indeed monitoring and reducing the app's CPU usage offers several benefits, including enhancing the user experience with quicker performance and extending battery life. The figure below illustrates the default view of the *CPU Profiler*, which includes several timelines:

- Event Timeline: displays the transitions of app activities through various lifecycle states and user interactions, like screen rotations.
- CPU Timeline: shows the real-time CPU usage of your app as a percentage of the total available CPU time and the number of threads used. It also compares the CPU usage of the app with other processes.
- Thread Activity Timeline: lists each thread of the app process and shows its activity using specific colors:
  - Green: the thread is active or ready to use the CPU, indicating a running or runnable state.
  - Yellow: the thread is active but waiting on an I/O operation, such as disk or network I/O.
  - Gray: the thread is sleeping and not consuming CPU time, often due to waiting for an unavailable resource, either voluntarily or because the kernel has put it to sleep until the resource is available.

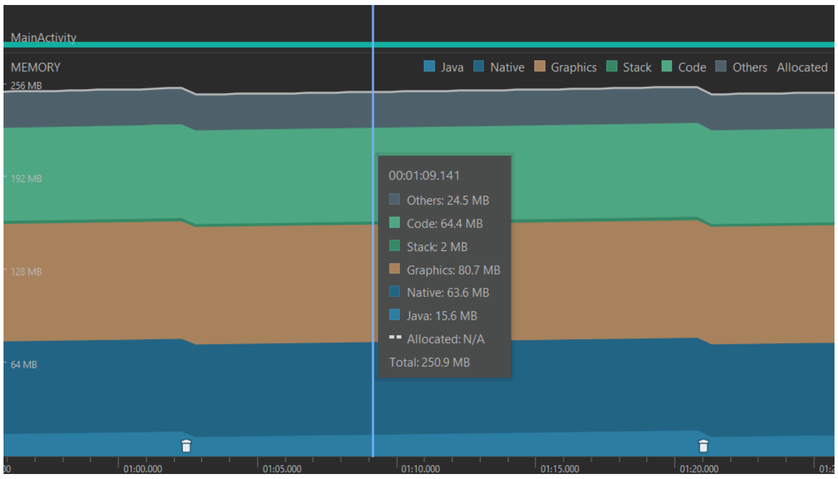
The *Memory Profiler* is a tool within the Android Profiler that assists in detecting memory leaks and excessive memory allocation, which can cause stuttering, freezing, or crashes in apps. It provides a real-time graph of your app's memory usage and allows you to capture a heap dump, initiate garbage collections, and monitor memory allocations.
The memory categories include:

- Java: Memory from objects created in Java or Kotlin code.
- Native: Memory from objects created in C or C++ code. This includes memory used by the Android framework for tasks like image and graphics processing, even if your app is entirely in Java or Kotlin.
- Graphics: Memory for graphics buffer queues, such as GL surfaces and textures, which are shared with the CPU, not exclusive to the GPU.
- Stack: Memory for native and Java stacks, linked to the number of active threads in the app.

- Code: Memory used for the app's code and resources, including dex bytecode, compiled dex code, .so libraries, and fonts.
- Others: Memory that the system cannot classify under any specific category.
- Allocated: The count of Java/Kotlin objects your app has allocated, excluding those in C or C++.



(a) peak at 10s from starting the app



(b) behaviour after 1 minute from the starting of the app

Figure 8.3: MEMORY profiler: app in OPENED state

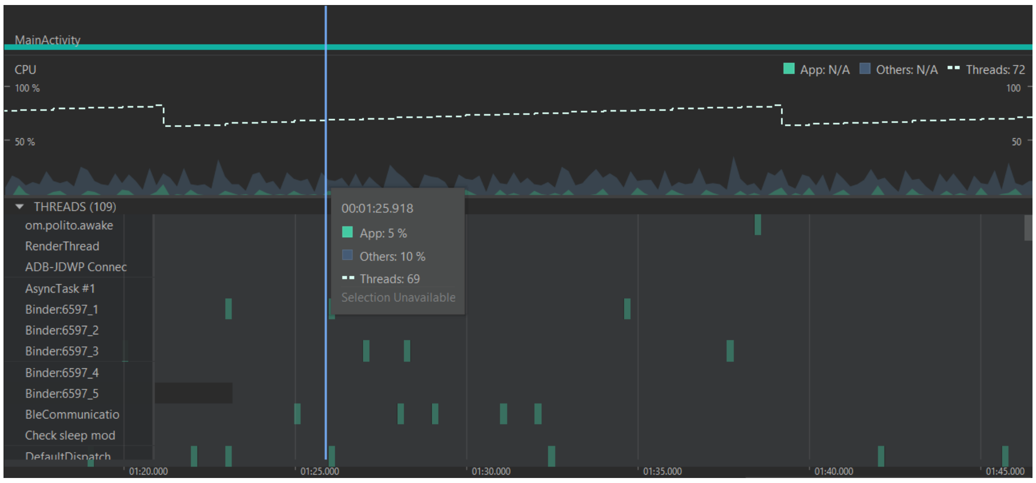
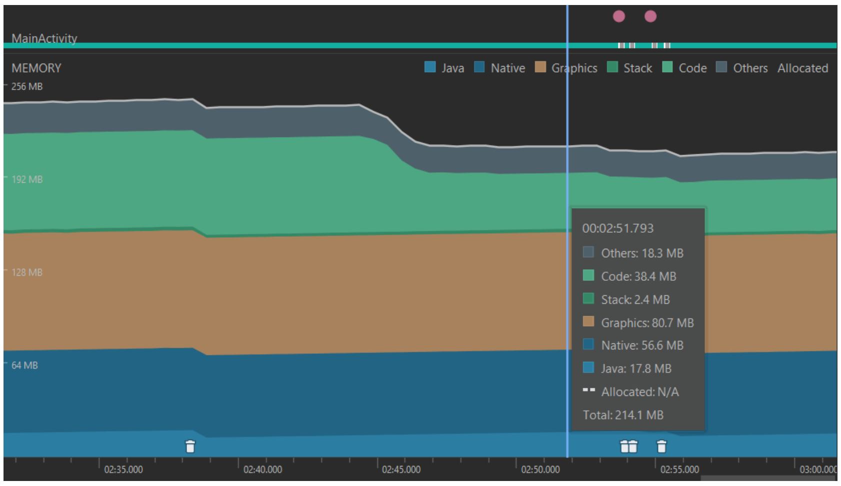
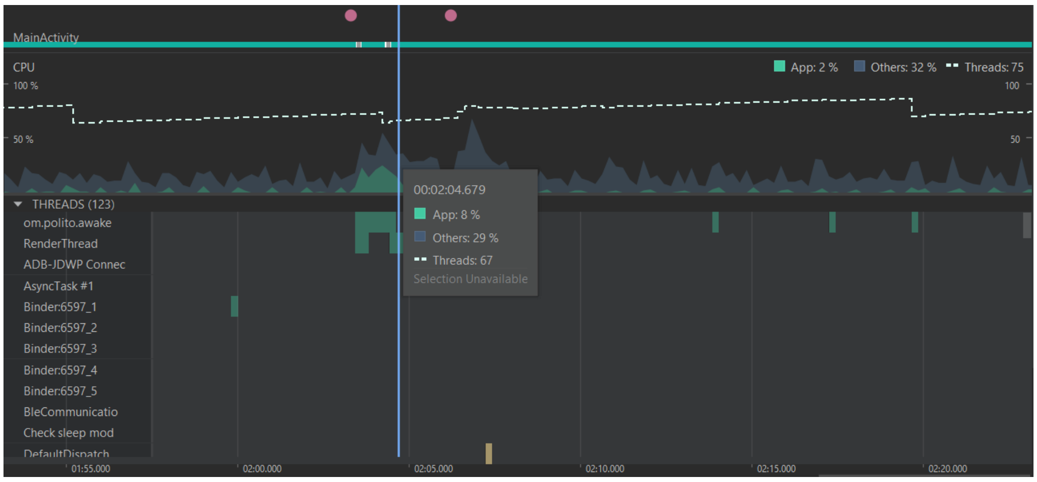


Figure 8.4: CPU profiler: app in OPENED state

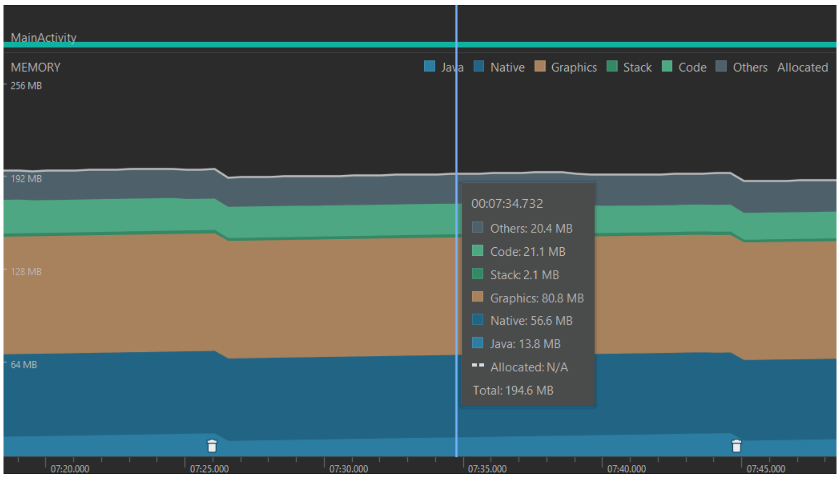


(a) MEMORY profiler

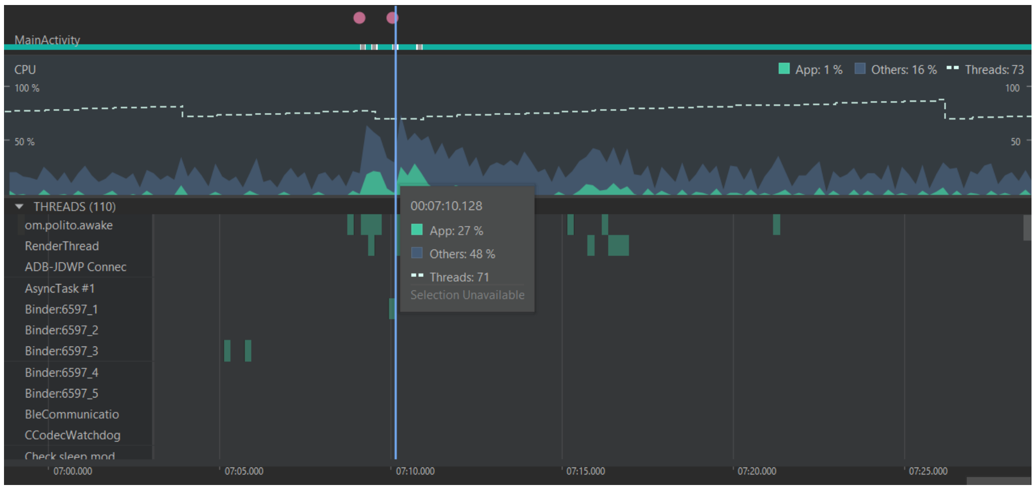


(b) CPU profiler

Figure 8.5: app in CALIBRATION state



(a) MEMORY profiler



(b) CPU profiler

Figure 8.6: app in AWAKE state

I've started the Profiler tool registration almost at the same time debugging the application on a smartphone, in such a way to monitor the changing of CPU and memory usages during a normal usage of the app implemented with NN.
In Fig 8.3a I reported the highest memory placement experienced after almost 10s from starting the app: 275MB. After this peak the memory profile stabilizes and tends to decrease gradually, but remaining above 194MB as in Fig 8.6a.

The memory usage of the app implemented with NN is high is compared to the one of smartphone commercial applications.

I didn't got access to the PredictS app Profiler, but considering that the generated apk for this particular NN is 477 MB, compared to the apk of last version of PredictS: 56.7M, I can easily derive that in terms of memory occupation this method is not convenient.

8.3.2 Battery consumption

I've mad a test comparing both algorithm: the one containing the RNN and the original PredictS, implemented in a Xiomi Redmi smartphone: I've started the app when the smartphone was 100% recharged and kept it running till 0% of battery and registered the incidence of both sw on the autonomy of the system.

algorithm	starting time	finishing time	overall time
RNN	20:47:46	10:06:41	14:41:05
PredictS	12:40:24	04:37:12	16:03:12

Table 8.2: app test info

The app implementing the RNN leads to a discharging of the smartphone around 1 hour, 22 minutes and 7 seconds before the app implementing PredictS sw.

Chapter 9

Conclusions

This thesis has demonstrated the potential of Recurrent Neural Networks (RNNs), particularly Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) networks, to replicate and potentially improve upon the proprietary PredictS algorithm developed by Sleep Advice Technologies for detecting driver drowsiness. The results indicate that these models can detect the onset of drowsiness several minutes in advance, providing critical insights into improving driver safety systems.

Key findings from the research include:

- **Model Performance**:
  The GRU-based model achieved notable performance metrics, with an accuracy of 85%, specificity of 97%, and precision and sensitivity above 70%. Despite the class imbalance present in the dataset, these results indicate the model's effectiveness in replicating the PredictS algorithm for drowsiness detection. It was also found that the models' sensitivity could be further improved by addressing the imbalance between the awake and drowsy labels.

- **Feature Selection**:
  The study confirmed that physiological signals, particularly heart rate (HR) and heart rate variability (HRV), were effective predictors for drowsiness. However, the introduction of additional features such as age introduced performance degradation, likely due to the limited and skewed age distribution in the dataset. This underscores the need for careful feature selection, particularly when dealing with smaller datasets.

- **Generalization and Dataset Expansion**:
  A significant challenge identified was the limited generalizability of the models to new datasets. The model's performance degraded when applied to additional datasets collected from real-world driving conditions, suggesting that expanding the dataset and ensuring a more diverse set of training data is essential for improving robustness. This would enhance the model's ability to adapt to various driving scenarios, environmental conditions, and physiological differences among drivers.

- **Algorithm Protection**:
  A novel contribution of this thesis is the introduction of a random jitter mechanism to protect the drowsiness detection algorithm from reverse engineering. This technique did not substantially impact model performance, making it a viable option

for companies looking to safeguard proprietary algorithms while maintaining their operational effectiveness.

- **Mobile Integration and Practical Application**:
  The successful integration of the GRU model into an Android application highlighted the feasibility of real-time drowsiness detection on mobile devices. However, high memory usage and significant battery consumption present hurdles for widespread adoption. Optimizations in model size and computational efficiency are required for better performance on mobile platforms, particularly for battery-dependent applications.

Future Work

The findings of this thesis suggest several avenues for future research:

- **Dataset Expansion**:
  Given the generalization challenges observed, future efforts should focus on acquiring more diverse datasets. Larger datasets that account for varying driving conditions, age groups, and physiological characteristics will be crucial in improving the robustness and reliability of the models.
- **Mobile Optimization**:
  Efforts to reduce the memory and battery consumption of the deployed models on mobile devices are necessary. Techniques such as model compression, quantization, and pruning could be explored to make the algorithms more suitable for real-time applications on smartphones or wearable devices.
- **Real-World Testing**:
  Further testing in real-world driving scenarios, beyond simulation environments, will be necessary to validate the models' effectiveness in practical applications. This would provide a more comprehensive understanding of the model's reliability and impact on improving driver safety.

In conclusion, this research contributes to the growing field of AI-based driver assistance systems, demonstrating the potential of RNNs in drowsiness detection. With further optimizations and dataset enhancements, the models developed in this study could form the basis for future applications aimed at reducing drowsiness-related accidents and enhancing road safety.

In summary, this research successfully demonstrated the potential of RNNs, particularly GRUs, to replicate the capabilities of the PredictS algorithm in detecting driver drowsiness, even if the dataset counted only 24 registrations, also characterized by unbalancing of data between labels.

The findings suggest that with further optimization—both in terms of model performance and mobile integration—this approach could play a significant role in enhancing driver safety. Future work should prioritize expanding the dataset and improving the model's efficiency for real-world deployment.

Bibliography

[1] et al. F. Lyu X. Zhang. «A Review of Active Automotive Safety Systems for Autonomous Vehicles». In: *IEEE Access* (2018).

[2] A. Smith J. Doe. «The History and Evolution of Driver Assistance Systems: From Cruise Control to Autonomous Driving». In: *Journal of Automotive Safety* (2020).

[3] M. T. Wong T. Peters. «Human-Centric Design of Advanced Driver Assistance Systems: The Role of Driver Monitoring». In: *International Journal of Vehicle Autonomous Systems* (2019).

[4] Shehzad Saleem. «Risk assessment of road traffic accidents related to sleepiness during driving: a systematic review». In: *East Mediterr Health J. 2022 Sep 29;28(9):695-700* (2022).

[5] Nyström B Carter N Ulfberg J and Edling C. «Sleep debt, sleepiness and accidents among males in the general population and male professional drivers.» In: *Accid Anal Prev. 2003 Jul;35(4):613-7.* (2003).

[6] Lagarde E Philip P Sagaspe P, Leger D, Ohayon MM, Bioulac B, Boussuge J, and Taillard J. «Sleep disorders and accidental risk in a large group of regular registered highway drivers.» In: *Sleep Med. Volume 11, Issue 10 , December 2010, Pages 973-979* (2010).

[7] Lesloum RH BaHammam AS Alkhunizan MA, Alshanqiti AM, Aldakhil AM, and Pandi-Perumal SR et al. «Prevalence of-related accidents among drivers in Saudi Arabia.» In: *Ann Thorac Med. 2014 Oct;9(4):236–41.* (2014).

[8] Barger LK Czeisler CA Wickwire EM, Dement WC, Gamble K, and et al. Hartenbaum N. «Sleep-deprived motor vehicle operators are unfit to drive: a multidisciplinary expert consensus statement on drowsy driving.» In: *Sleep Health, 2016 Jun;2(2):94–9.* (2016).

[9] L. N. Zhang A. K. Gupta. «Heart Rate Variability as a Predictor of Drowsiness: An Application for Driver Assistance Systems». In: *Journal of Applied Physiology* (2019).

[10] M. S. Taylor D. Vural A. Bordallo. «Real-Time Detection of Driver Drowsiness Using Vehicle-Based Measures». In: *IEEE Electrical Insulation Magazine (Volume: 37, Issue: 3, May/June 2021)* (2021).

[11] Yi Lu Murphey Jiaqi Ma and Hong Zhao. «Real time drowsiness detection based on lateral distance using wavelet transform and neural network, pages 411–418». In: *Institute of Electrical and Electronics Engineers Inc.* (2015).

[12] Renjie Li Zuojin Li Shengbo Eben Li, Bo Cheng, and Jinliang Shi. «Online detection of driver fatigue using steering wheel angles for real driving conditions». In: *Sensors (Switzerland)* (17 march 2017).

[13] H. Fujimoto M. Hiraiwa T. Yamashita. «Driver Drowsiness Detection Based on Eye Closure and Head Tilting». In: *2020 IEEE Global Engineering Education Conference (EDUCON), Porto, Portugal, 2020, pp. 1217-1224* (2020).

[14] M. Sotelo M. Bergasa J. Nuevo. «Face and Eye Tracking for Driver Monitoring». In: *IEEE Transactions on Haptics, vol. 14, no. 3, pp. 577-590, 1 July-Sept. 2021* (2021).

[15] Mohammed Awad Yaman Albadawi Maen Takruri. «A review of recent developments in driver drowsiness detection systems». In: *Sensors* (2022).

[16] Hamidreza Bakhoda Serajeddin Ebrahimian Hadi Kiashari Ali Nahvi, Amirhossein Homayounfard, and Masoumeh Tashakori. «Evaluation of driver drowsiness using respiration analysis by thermal imaging on a driving simulator». In: *Multimedia Tools and Applications* (2020).

[17] Shelby E McDonald Douglas M Wiegand Julie A McClafferty and Richard J Hanowski. *Development and evaluation of a naturalistic observer rating of drowsiness protocol.* National Surface Transportation Safety Center for Excellence, 2009.

[18] João Mateus Marques Santana Caio Bezerra Souto Maior Márcio José das Chagas Moura and Isis Didier Lins. «Real-time classification for autonomous drowsiness detection using eye aspect ratio». In: *Expert Systems with Applications* (2020).

[19] Nur Syazarin Natasha Abd Aziz Ameen Aliu Bamidele Kamilia Kamardin, Suriani Mohd Sam, Irfanuddin Shafi Ahmed, Azizul Azizan, Nurul Aini Bani, and Hazilah Mad Kaidi. «Non-intrusive driver drowsiness detection based on face and eye tracking». In: *International Journal of Advanced Computer Science and Applications* (2019).

[20] Laura Prest Ali Shahidi Zandi Azhar Quddus and Felix JE Comeau. «Nonintrusive detection of drowsy driving based on eye tracking data». In: *Transportation research record* (2019).

[21] Marley Vellasco Alimed Celecia Karla Figueiredo and René González. «A portable fuzzy driver drowsiness estimation». In: *Sensors* (2020).

[22] M. A. Lima P. A. Castro R. A. Mendez. «Heart Rate Variability and Electrodermal Activity as Indicators of Driver Drowsiness». In: *Monthly Notices of the Royal Astronomical Society, vol. 492, no. 1, pp. 5121-5140* (Oct, 2019).

[23] P. Martinez J. F. Ochoa. «Electroencephalogram (EEG) Based Driver Drowsiness Detection». In: *OCEANS 2019 - Marseille, Marseille, France, 2019, pp. 1-6* (2019).

[24] J. Lee H. Lee and M. Shin. «Using wearable ecg/ppg sensors for driver drowsiness detection based on distinguishable pattern of recurrence plots». In: *Electronics* (2019).

[25] J.-i. Lee S. Koh B.R. Cho, S.-O. Kwon, S. Lee, J.B. Lim, S.B. Lee, and H.-D. Kweon. «Driver drowsiness detection via ppg biosignals by using multimodal head support». In: *In 2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)* (2017).

[26] N. Sofra T. Kundinger and A. Riener. «Assessment of the potential of wristworn wearable sensors for driver drowsiness detection». In: *Sensors* (2020).

[27] K. Kamata K. Fujiwara E. Abe, C. Nakayama, Y. Suzuki, T. Yamakawa, T. Hiraoka, M. Kano, Y. Sumi, and F. Masuda. «Heart rate variability-based driver drowsiness detection and its validation with eeg». In: *IEEE Transactions on Biomedical Engineering* (2018).

[28] J. Ramos-Castro F. Guede-Fernandez M. Fernandez-Chimeno and M.A. Garcia-Gonzalez. «Driver drowsiness detection based on respiratory signal analysis». In: *IEEE Access* (2019).

[29] R. Smith K. M. Peters. «A Hybrid Model for Driver Drowsiness Detection Using Vehicle Dynamics and Physiological Signals». In: *IEEE Access, vol. 9, pp. 76493-76502* (2021).

[30] D. Zhang L. Wang S. Zhai. «Multi-Modal Driver Fatigue Detection: A Review of Hybrid Approaches». In: *IEEE Internet of Things Journal, vol. 9, no. 13, pp. 10936-10947* (1 July 2022).

[31] H. Su and G. Zheng. «A partial least squares regression-based fusion model for predicting the trend in drowsiness». In: *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans, vol. 38, no. 5, pp. 1085–1092* (2008).

[32] A. Hashemi et al. «Time driver's drowsiness detection by processing the eeg signals stimulated with external flickering light». In: *Basic Clin. Neurosci., vol. 5, no. 1, pp. 22–27* (2014).

[33] R. O. Mbouna et al. «Visual analysis of eye state and head pose for driver alertness monitoring». In: *IEEE Trans. Intell. Transp. Syst., vol. 14, no. 3, pp. 1462–1469* (Sep. 2013).

[34] A. J. Camm et al. «Guidelines heart rate variability—standards of measurement, physiological interpretation, and clinical use». In: *Eur. Heart J., vol. 115, no. 5, pp. 354–381* (1996).

[35] F. Versace et al. «Heart rate variability during sleep as a function of the sleep cycle». In: *Biol. Psychol., vol. 63, no. 2, pp. 146–162* (2003).

[36] Hamdy R. M., Abdel-Tawab H., Abd Elaziz O. H., Sobhy El attar R., and Kotb F. M. «Evaluation of Heart Rate Variability Parameters During Awake and Sleep in Refractory and Controlled Epileptic Patients». In: *International Journal of General Medicine* 15 (2022), pp. 3865–3877.

[37] Aymen A. Alian and Kirk H. Shelley. «Photoplethysmography. Best Practice Research Clinical Anaesthesiology». In: (2014).

[38] Luigi Pugliese. «PhD. Algorithm for the detection of the drowsiness». PhD thesis. 2024.

[39] Riccardo Groppo Luigi Pugliese Massimo Violante. «Real-time Sleep Prediction Algorithm using Commercial Off the Shelf Wearable Devices». In: *2023 IEEE Smart World Congress (SWC)* (2023).

[40] IBM. *Supervised learning.* https://www.ibm.com/topics/supervised-learning., [data di pubblicazione non disponibile]. Consultato il 24 luglio 2024.

[41] Göran Kecklund Anna Åkerstedt Miley and Torbjörn Åkerstedt. «Comparing two versions of the karolinska sleepiness scale (kss). Sleep and biological rhythms». In: *14:257–260* (2016).

[42] Ning Wang Wen Zhu Nancy Seng. «Sensitivity, specificity, accuracy, associated confidence interval and roc analysis with practical sas implementations». In: *NESUG proceedings: health care and life sciences, Baltimore, Maryland, 19:67* (2010).

[43] Eric Rothstein Morris Ralf C. Staudemeyer. «Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks». In: *arXiv:1909.09586* (12 September 2019).

[44] Livingstone DJ. *Artificial neural networks: methods and applications.* 2008.

[45] Khosro Sadeghniiat Haghighi MD PhD3 Zeinab Kohzadi MSc1 Reza Safdari PhD2. «Determination of Sleep Apnea Severity Using Multi-Layer Perceptron Neural Network». In: *The Korean Society of Sleep Medicine* (2020).

[46] Paul J. Werbos. «Backpropagation through time: What it does and how to do it.» In: *Proc. of the IEEE, 78(10):1550–1560* (1990).

[47] Ronald J. Williams and David Zipser. «A learning algorithm for continually running fully recurrent neural networks.» In: *Neural Computation, 1(2):270–280* (jun 1989).

[48] Michael C Mozer. «Induction of Multiscale Temporal Structure.» In: *Advances in Neural Information Processing Systems 4, pages 275–282. Morgan Kaufmann* (1992).

[49] Sepp Hochreiter and J¨urgen Schmidhuber. *Long Short-Term Memory. Neural computation.* 1997.

[50] Yoshua Bengio Junyoung Chung Caglar Gulcehre KyungHyun Cho. «Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling». In: *arXiv:1412.3555v1 [cs.NE]* (2014).

[51] Zachary Kirori and Edwin Ireri. «Towards optimization of the gated recurrent unit (GRU) for regression modeling». In: *International Journal of Social Sciences and Information Technology* (2020).

[52] Huaizheng Lu by Xinyi Wu Bingjie Xiang, Chaopeng Li, Xingwang Huang, and Weifang Huang. «Optimizing Recurrent Neural Networks: A Study on Gradient Normalization of Weights for Enhanced Training Efficiency». In: *Applied Sciences* (2014).

[53] John Peurifoy1 Li Jing1 Caglar Gulcehre, Yichen Shen1, Max Tegmark1, Marin Soljačić1, and Yoshua Bengio. «Gated Orthogonal Recurrent Units: On Learning to Forget». In: *arXiv:1706.02761* (2017).

[54] Xinye Chen Roberto Cahuantzi and Stefan Güttel. «A comparison of LSTM and GRU networks for learning symbolic sequences». In: *arXiv:2107.02248* (2023).