Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Informationstechnik
(Schwerpunkt Kommunikationselektronik)

LIKE

Master's Thesis

Evaluation of Traditional and Machine Learning Algorithms for Real-time Parkinson's Tremor Detection

Author	Ege Özkoc
Matriculation No.	23230154
Course of Study	Medical Engineering
Supervisor	Prof. Dr.-Ing. Albert Heuberger
	Dr. rer. nat. Tobias Sebastian Zech
	M. Sc. Norman Pfeiffer
	Dipl.-Ing. Jürgen Frickel
	PD Dr. phil. Heiko Gaßner
Registration Date	27 May 2024
Submission Date	13 November 2024

Declaration of Originality

I, Ege Özkoc (23230154), hereby confirm that I completed the submitted work independently and without the unauthorized assistance of third parties and without the use of undisclosed and, in particular, unauthorized aids. This work has not been previously submitted in its current form or in a similar form to any other examination authorities and has not been accepted as part of an examination by any other examination authority.

Where the wording has been taken from other people's work or ideas, this has been properly acknowledged and referenced. This also applies to drawings, sketches, diagrams and sources from the Internet.

In particular, I am aware that the use of artificial intelligence is forbidden unless its use an aid has been expressly permitted by the examiner. This applies in particular to chatbots (especially ChatGPT) and such programs in general that can complete the tasks of the examination or parts thereof on my behalf.

Furthermore, I am aware that working with others in one room or by means of social media represents the unauthorized assistance of third parties within the above meaning, if group work is not expressly permitted. Each exchange of information with others during the examination, with the exception of examiners and invigilators, about the structure or contents of the examination or any other information such as sources is not permitted. The same applies to attempts to do so.

Any infringements of the above rules constitute fraud or attempted fraud and shall lead to the examination being graded "fail" ("nicht bestanden").

Erlangen, 13.11.2024_____

Acknowledgement

I would like to sincerely thank my supervisor, Dipl.-Ing. Jürgen Frickel, for his ongoing support, valuable feedback, and guidance throughout my thesis. His expertise and commitment have greatly shaped this work, and I am very grateful for his patience and encouragement along the way. I am also deeply thankful to Dr. rer. nat. Tobias Sebastian Zech, whose knowledge and dedication made our discussions on machine learning in biomedical engineering both insightful and productive. His help with complex problems and willingness to share his expertise were essential in overcoming many challenges. I would like to extend my gratitude to Prof. Dr.-Ing. Albert Heuberger and M. Sc. Norman Pfeiffer for giving me the opportunity to work on this thesis at Fraunhofer IIS and supervising this work. Their support made this experience possible and greatly enriched my research. I am also thankful to PD Dr. phil. Heiko Gaßner for reviewing and supporting the medically relevant aspects of this thesis. His insights were invaluable in enhancing the quality of this work.

I also want to express my heartfelt thanks to my wife, Meltem Sahin Ozkoc, for her constant support, understanding, and encouragement during the challenging times of this thesis. Her belief in me gave me the motivation to keep going. I am also grateful to my parents for their endless love and encouragement, which have been a foundation throughout my academic journey. Their faith in me and support have inspired me to reach this important milestone. This achievement would not have been possible without the collective support, guidance, and inspiration from each of these people.

Topic and Task of this Thesis

Topic:

Parkinson's disease is a neurodegenerative disorder that affects over 8.5 million people all around the world, and severely affect their quality of life. Hand tremors are the primary symptom of the disease. There have been previous research on Parkinson's tremor detection, where the focus was mainly on monitoring the progression of Parkinson's disease, without much consideration of real-time detection of Parkinson's disease tremors on embedded devices. However, to develop wearable devices that can detect and mitigate Parkinson's tremor, and thereby increase the quality of life for Parkinson's patients, we need real-time solutions for embedded devices that can detect tremors with high accuracy, energy efficiency, and low latency. There is a gap in the literature, since existing literature does not focus on real-time detection of tremors on embedded devices. This thesis focuses on evaluating both traditional signal processing techniques and novel machine learning algorithms to accurately and efficiently detect tremors, with particular focus on problems that come with embedded device implementation and optimization of the machine learning algorithms to enable inference on embedded devices.

Task:

In this thesis, an extensive literature review is done both on Parkinson's tremor detection and Edge AI. Then the selection of algorithms to detect Parkinson's tremor are presented and the details of the implemented algorithms are given. After that, the evaluation results of the algorithms are presented and the results are discussed to understand which algorithms are more suitable for Parkinson's tremor detection on embedded devices. Additionally, effect of neural network architecture on trade-offs between model size, model performance and energy efficiency are discussed. Furthermore, extensive experiments are done on model optimization techniques such as pruning and quantization of neural network models, and the effect of these techniques on the model performance are discussed. Finally, future directions to enhance Parkinson's tremor detection in real-time on embedded devices are proposed.

Abstract

Parkinson's disease is a progressive neurodegenerative disorder that affects more than 8.5 million people worldwide, significantly impacting the quality of life of patients through symptoms such as hand tremors. Even though existing research has explored tremor detection, most of the existing literature does not consider real-time, embedded solutions for Parkinson's disease tremor detection, which is critical to develop wearable devices that can detect and mitigate Parkinson's disease tremors. This thesis addresses this gap by evaluating both traditional signal processing methods and advanced machine learning algorithms for real-time Parkinson's tremor detection on resource-constrained embedded devices.

We implement and evaluate various algorithms, from more traditional ones to more novel ones, including Fourier transform based methods, Support Vector Machines, Random Forests, and Convolutional Neural Networks to achieve accurate, low-latency, and energy efficient Parkinson's tremor detection. In the experiments we examine the performance of the implemented algorithms in terms of accuracy, latency, and energy efficiency, with a particular focus on optimization techniques like pruning and quantization. Quantitative results are presented to understand the trade-offs in accuracy, model complexity, decision latency, and computational efficiency, with CNNs optimized through quantization and pruning demonstrating promising results for real-time, edge-based deployment.

Our findings contribute valuable insights for developing energy-efficient, real-time tremor detection systems that can be embedded in wearable devices, facilitating both mitigation of Parkinson's disease hand tremors and continuous patient monitoring. Future work should investigate further optimizations and clinical testing to enhance deployment viability and accuracy.

Kurzzusammenfassung

Die Parkinson-Krankheit ist eine fortschreitende neurodegenerative Erkrankung, die weltweit Millionen von Menschen betrifft und die Lebensqualität der Patienten durch Symptome wie Zittern der Hände (Tremor) erheblich beeinträchtigt. Obwohl sich die vorhandene Forschung mit der Tremorerkennung befasst hat, werden in der Literatur größtenteils keine eingebetteten Echtzeitlösungen zur Erkennung von Parkinson-Tremor berücksichtigt, die für die Entwicklung tragbarer Geräte, die Parkinson-Tremor erkennen und abschwächen können, von entscheidender Bedeutung sind. Diese Arbeit schließt diese Lücke, indem sie sowohl traditionelle Signalverarbeitungsmethoden als auch fortschrittliche maschinelle Lernalgorithmen zur Echtzeit-Erkennung von Parkinson-Tremor auf eingebetteten Geräten mit eingeschränkten Ressourcen bewertet.

Wir implementieren und bewerten verschiedene Algorithmen, von traditionelleren bis hin zu neuartigeren, darunter Fourier transformation basierte Methoden, Support Vector Machines (SVMs), Random Forests und Convolutional Neural Networks (CNNs), um eine genaue, latenzarme und energieeffiziente Parkinson-Tremorerkennung zu erreichen. In den Experimenten untersuchen wir die Leistung der implementierten Algorithmen in Bezug auf Genauigkeit, Latenz und Energieeffizienz, wobei wir uns insbesondere auf Optimierungstechniken wie Beschneiden und Quantisierung konzentrieren. Quantitative Ergebnisse werden präsentiert, um die Kompromisse bei Genauigkeit, Modellkomplexität, Entscheidungslatenz und Rechenleistung zu verstehen, wobei durch Quantisierung und Beschneiden optimierte CNNs vielversprechende Ergebnisse für die Echtzeit-Bereitstellung auf Edge-Basis liefern.

Die Ergebnisse dieser Arbeit liefern wertvolle Erkenntnisse für die Entwicklung energieeffizienter Echtzeit-Tremorerkennungssysteme, die in tragbare Geräte eingebettet werden können und sowohl die Linderung von Handtremor bei Parkinson-Krankheit als auch die kontinuierliche Patientenüberwachung ermöglichen. Zukünftige Arbeiten sollten weitere Optimierungen und klinische Tests untersuchen, um die Einsatzfähigkeit und Genauigkeit zu verbessern.

Contents

1 Introduction

1.1 Motivation

Parkinson's disease is a progressive neurodegenerative disease that affects 2-3% of the population older than 65 years of age, which makes it the second-most common neurodegenerative disorder [1]. According to the World Health Organization, more than 8.5 million people worldwide are affected by the disease [2]. PD is mainly characterized by cardinal motor symptoms such as bradykinesia, rigidity, and resting tremor [3]. Hand tremor is one of the disease's most common symptoms, affecting the quality of life of PD patients [4]. For various reasons, it is crucial to detect tremors accurately using Inertial Measurement Unit (IMU) data obtained from wearable devices. First is to monitor the progression of the disease remotely, which enables tracking the condition of the PD patient outside the clinical environment, second is to adjust the treatment plans, such as adjusting the dosage of medication in response to the severity of tremors, and last but not least to develop wearable devices such as gloves that can detect the tremors in real-time and then mitigate it, improving the quality of life for PD patients.

The existing tremor detection systems are mostly being used in clinical environments with special equipment. While there has been some research on Parkinson's tremor detection, on developing algorithms to accurately detect Parkinson's tremor for remote monitoring of the patients outside the hospital/lab environment [5], there is limited research on real-time Parkinson's tremor detection on edge devices, which is crucial in developing wearable devices, such as gloves, that can detect and mitigate tremors. The existing research mostly aims to detect tremors after recording the IMU data and computing the algorithms on cloud [6]. As a result, these approaches do not give much consideration to key factors, such as model complexity, decision latency, or computational efficiency, which are critical for deployment on edge devices.

This thesis aims to focus on real-time Parkinson's tremor detection on edge devices, which is crucial in designing devices that could directly improve the quality of life of PD patients. Specifically, this thesis focuses on designing both traditional and machine learning models that could be deployed on embedded devices that are both accurate and computationally

efficient. The ultimate goal is to enable real-time detection of Parkinson's disease hand tremors on edge devices and mitigation of Parkinson's tremors, thereby directly enhancing the quality of life for PD patients.

1.2 Outline of Thesis

This thesis is organized into the following chapters:

- **Background and Theory**
  In this chapter, we briefly provide an overview of Parkinson's disease and shortly give the underlying physiological mechanisms of its motor symptoms, particularly hand tremors. Then, we review the existing literature on detecting Parkinson's tremors, both inside and outside the clinical environment. Additionally, we give the mathematical and theoretical background of the algorithms that will be used in the following chapters.
- **Literature Review on Parkinson's Tremor Detection and Edge AI**
  In this chapter, an extensive literature review is done on Parkinson's tremor detection and the implementation of machine learning models on embedded devices, known as edge AI.
- **Methods**
  In this chapter, we describe the dataset that we used to train and evaluate the ML algorithms by describing where we got the data and how the IMU data was acquired in the study. Then, we discuss how the data is preprocessed, and we provide the implementation details of each of the traditional and ML algorithms that we used.
- **Evaluation of the Implemented Algorithms and Optimization Methods**
  In this chapter, we present the results that are obtained from the evaluation of the implemented algorithms. We compare the results using various performance metrics, including but not limited to accuracy, precision, recall, and F1-score. We discuss the trade-offs between the model complexity and performance, which are crucial for real-time implementation on edge devices. We investigate the effect of different CNN architectures on the trade-offs between model size, model complexity, and model performance. We further experiment with neural network optimization methods, such as quantization of weights and neural network pruning, and investigate the effects of these techniques on the model performance.

- **Discussion and Conclusion**
  In this chapter, the evaluation results and the trade-offs between the implemented algorithms are discussed. The most suitable algorithms for real-time Parkinson's tremor detection on embedded devices are proposed.

- **Future Work**
  This thesis concludes with a discussion of how the proposed methods could be improved further, such as further optimizing models for better accuracy, computational efficiency, and lower latency. Furthermore, a future clinical trial for evaluating the implemented algorithms in the real world with Parkinson's patients is proposed.

2 Background and Theory

In this chapter, we explore the underlying physiological reasons for Parkinson's disease and the basis of tremor symptoms. Additionally, we examine both the traditional signal processing techniques and machine learning algorithms that we implement and evaluate. We further present neural network optimization and regularization methods that we use in our model implementation. Finally, we review edge artificial intelligence and tiny machine learning to implement machine learning models on resource-constrained edge devices.

2.1 Parkinson's Disease and Tremor

Parkinson's Disease (PD) is a complex progressive neurodegenerative disease that is mainly characterized by the cardinal motor symptoms such as bradykinesia, rigidity, and resting tremor [3], as depicted in Figure 2.1. PD is the most common neurodegenerative disease after Alzheimer's disease [7], and it is estimated that more than 8.5 million people are living with PD worldwide, according to the World Health Organization [2]. The prevalence of the disease increases with age [8], [9] and around 2-3% of people above 65 years of age are living with that disease [1]. With the aging population, it is expected that the prevalence and incidence of PD among the population will increase in the future [8]. One of the most common symptoms of the disease is hand tremors, which affect up to 75% of the patients [10] and can be the most troublesome motor symptom for the patients, as it affects numerous activities of daily living such as dressing, eating, and drinking. [11]. While people with PD may experience several forms of tremor, the most common type of tremor is the resting tremor, and it occurs when the patient is at rest with the typical frequency of 4-6Hz [12] as depicted in Figure 2.2.

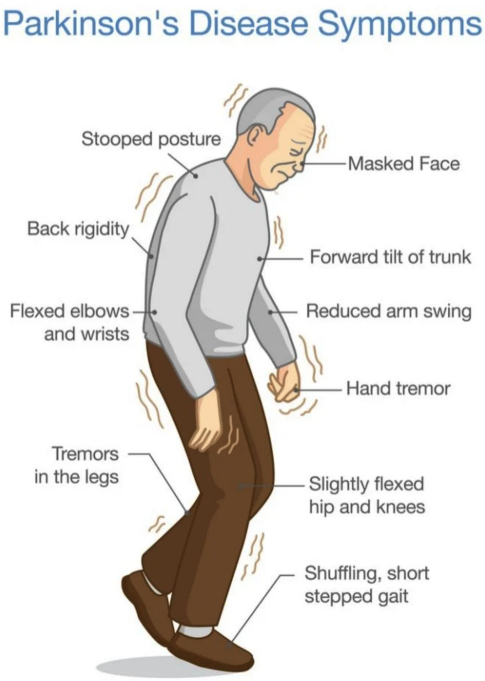

Figure 2.1: Parkinson's disease symptoms [13].

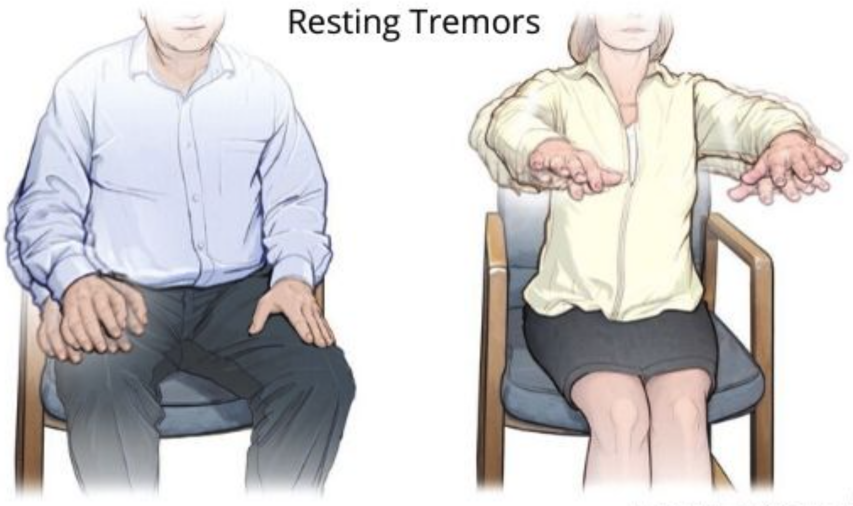

Figure 2.2: Parkinson's disease resting tremors [14].

The physiological reasons underlying Parkinson's disease can be explained by two distinct neural circuits in the brain: the basal ganglia and the cerebello-thalamo-cortical circuit.

According to the *"dimmer-switch model"* as depicted in Figure 2.3, tremor happens due to the interaction between these two circuits. Basal ganglia is where the tremor is triggered, and it can be considered as the *"switch"* for the PD tremor. When dopamine is depleted in the basal ganglia, specifically in the pallidum, due to the degeneration of dopaminergic neurons in the retrorubral area (A8 region), this leads to abnormal activity. Even though this does not cause PD tremor directly, it is the primary reason of it, as it triggers the start of the PD tremor. Another circuit critical for understanding the PD tremors is the cerebello-thalamo-cortical circuit. This circuit consists of the cerebellum, thalamus, and motor cortex, and it determines the amplitude of the tremor. It is the *"dimmer"* part of the model as it controls the intensity of the tremor after it is triggered by the basal ganglia as described. The motor cortex is the place where the two circuits, the basal ganglia, and the cerebello-thalamo-cortical circuit, interact with each other. To summarize, the basal ganglia acts as the *"switch"* and triggers the start of the tremor; whereas the cerebello-thalamo-cortical circuit acts as the *"dimmer"* and controls the amplitude of the tremor once it is triggered by the basal ganglia. [15]

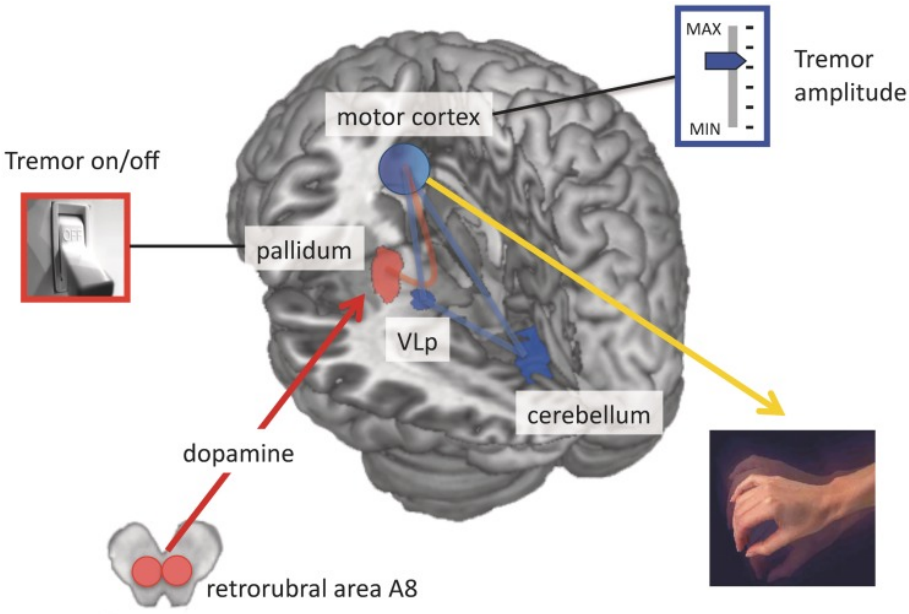

Figure 2.3: Dimmer switch model for PD tremors [15].

Another explanation of the underlying cause of the PD tremor is the dysfunction of neurotransmitters, such as dopamine, serotonin, noradrenaline, and acetylcholine. The most relevant neurotransmitter for PD tremors is dopamine, which is a neurotransmitter

that plays a critical role in the brain's ability to control human movement. In PD, there is a reduction in dopamine levels, and this results in unusual signaling between the basal ganglia and motor cortex, which leads to impaired motor control and tremors [10].

2.2 Signal Processing Algorithms

In this section, we review the fundamental transforms and algorithms of signal processing, such as the Continuous Time Fourier Transform (CTFT), the Discrete Fourier Transform (DFT), and the Fast Fourier Transform (FFT).

Continuous Time Fourier Transform

The Continuous Time Fourier transform (CTFT) is one of the fundamental transforms in signal processing that is used to analyze the frequency components in a continuous-time signal. It is mathematically described in Equation 2.2.1.

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t}\, dt \tag{2.2.1}$$

where $F(\omega)$ is the Fourier Transform of $f(t)$, ω is the angular frequency, and j is the imaginary unit [16].

The inverse Fourier Transform allows us to reconstruct the original time-domain signal from its frequency-domain representation:

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{j\omega t}\, d\omega \tag{2.2.2}$$

The CTFT cannot be used in digital systems where all the sequences are discrete. We need a transform that can transform a discrete signal into its discrete frequency representation, which will be presented in the following section.

Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is a signal processing technique to transform discrete time or spatial domain data into its discrete frequency domain representation. This transformation is particularly important in electrical engineering since it allows the frequency domain analysis of discrete signals on digital systems [17].

The DFT of a sequence of N complex numbers $x_0, x_1, \ldots, x_{N-1}$ is another sequence of N complex numbers $X_0, X_1, \ldots, X_{N-1}$ given as in Equation 2.2.3:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \ldots, N-1 \tag{2.2.3}$$

where j is the imaginary number ($j^2 = -1$), and k represents the index of the discrete frequency domain representation [17].

The inverse DFT (IDFT) is defined as:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j\frac{2\pi}{N}kn}, \quad n = 0, 1, \ldots, N-1 \tag{2.2.4}$$

This transformation allows for the reconstruction of the original signal from its discrete frequency domain representation [17].

Fast Fourier Transform (FFT)

Computing the DFT directly using the mathematical definition is computationally expensive, as it requires $O(N^2)$ operations. In practice, the Fast Fourier Transform algorithm is used to compute the DFT, which gives the same result as DFT; however, FFT is computationally much more efficient as it requires $O(N \log N)$ operations, making it practical to compute DFTs [18].

The FFT algorithm makes use of the symmetries and periodicities in the DFT calculation to avoid redundant computations. The way this is accomplished is by dividing N-point FFT calculation into 2 N/2-point FFT calculations. If N is a power of 2, this can be done recursively until there are only 2-point FFT calculations left, significantly reducing the

computational load. How an FFT calculations can be divided into smaller FFT calculations can be understood from the following equations:

$$X[k] = X_{\text{even}}[k] + e^{-j\frac{2\pi}{N}k} X_{\text{odd}}[k] \tag{2.2.5}$$

$$X[k+N/2] = X_{\text{even}}[k] - e^{-j\frac{2\pi}{N}k} X_{\text{odd}}[k] \tag{2.2.6}$$

Here, $X_{\text{even}}[k]$ and $X_{\text{odd}}[k]$ represent the DFTs of the even and odd-indexed elements of the input sequence, respectively. There is a symmetry between these two equations. Except for the sign of the odd component, the equations are the same, which can be used to reduce the computational load of the FFT calculations.

After computing 2-point FFTs, the combination of the intermediate results can be done using butterfly operations. [19] These operations consist of summing and subtracting terms in a structured way that makes use of the symmetries that are described. Each butterfly operation also involves multiplying by complex exponential terms. The diagram of the butterfly operations is shown in Figure 2.4.

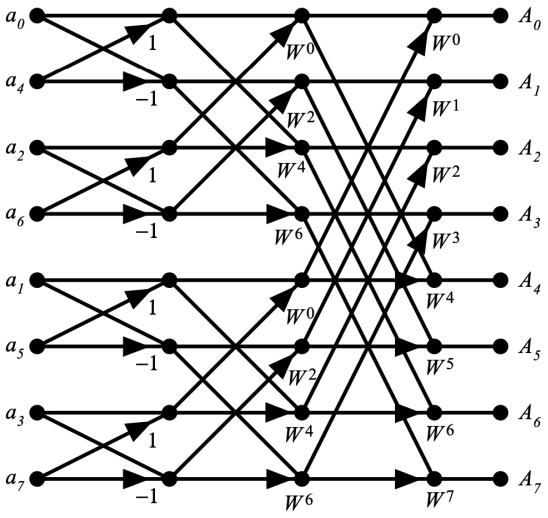

Figure 2.4: Butterfly operations [19].

The complex exponential terms are called twiddle factors and are computed as follows:

$$W_N^k = e^{-j\frac{2\pi k}{N}}$$

where:

- N is the total number of points in the DFT,
- k is the index of the current element
- $e^{-j\frac{2\pi k}{N}}$ is the complex exponential function, with j being the imaginary unit.

By computing the magnitude of FFT of accelerometer measurements for non-tremor and tremor movements, we can directly observe the difference in dominant frequencies for each of the movements. In Figures 2.5 and 2.6 we see that the dominant frequency for a non-tremor movement is around 1-2 Hz, whereas the dominant frequency for a tremor movement is around 5-6 Hz. Therefore FFT can be helpful in differentiating the non-tremor movement with tremor movement, either simply by finding the peak of the magnitude of FFT and applying a threshold to it, or combining FFT with more complex algorithms, where the frequency domain signal could be used as an input to another algorithm, which distinguishes non-tremor movement and tremor movement.

In this thesis, the implemented Fourier transform-based algorithms make use of the FFT algorithm due to its computational efficiency.

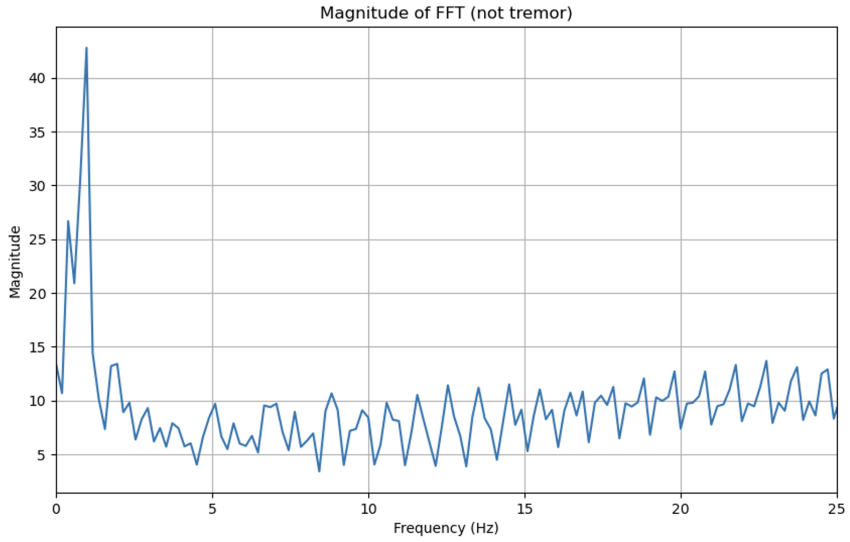

Figure 2.5: Magnitude of FFT of an accelerometer recording of a non-tremor movement.

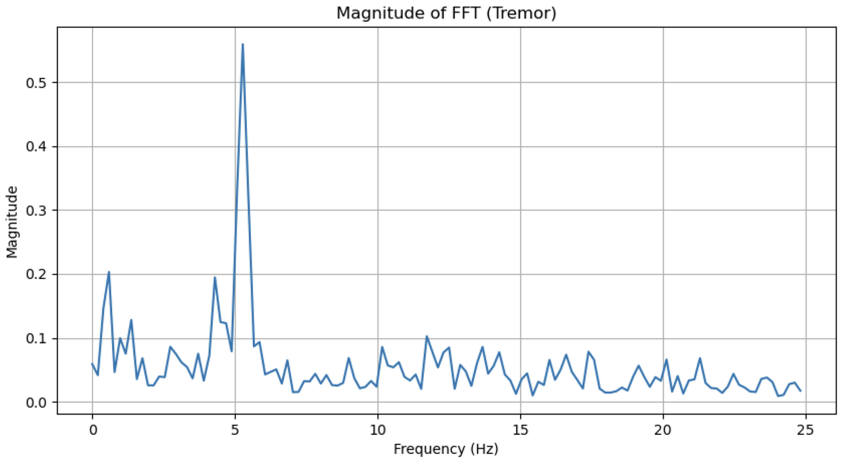

Figure 2.6: Magnitude of FFT of an accelerometer recording of a tremor movement.

2.3 Welch Method for Power Spectral Density Estimation

The Power Spectral Density (PSD) is a fundamental tool in signal processing, used to characterize the distribution of power of a signal over frequency. One of the most commonly used methods for estimating the PSD is the Welch method, introduced by Peter Welch in 1967 as an improvement over the standard periodogram approach [20].

The Welch method reduces the variance of the PSD estimate by averaging modified periodograms. In comparison to the basic periodogram, the Welch method results in a more stable and less noisy PSD estimate. The key idea behind the Welch method is the use of overlapping segments of the data, each of which is windowed before computing the periodogram. The periodograms are then averaged to obtain the final PSD estimate.

The Welch method involves the following key steps:

1. **Segmentation:** The input signal is divided into overlapping segments of equal length. The degree of overlap, typically 50%, is chosen to balance between variance reduction and computational efficiency. Let the signal be denoted by $x[n]$, and it is divided into M overlapping segments, each of length L, with an overlap of K samples.

2. **Windowing:** A window function $w[n]$, such as a Hamming or Hann window, is applied to each segment to reduce the spectral leakage that arises from finite-length

signal segments. Windowing ensures that the signal edges are smoothed before computing the periodogram. The windowed segment for the i-th segment is given by:

$$x_i[n] = w[n]x_i[n], \quad n = 0, 1, \ldots, L-1$$

3. **Periodogram Computation:** For each windowed segment, the periodogram is computed using the Discrete Fourier Transform (DFT). The periodogram for the i-th segment is given by:

$$P_i(f) = \frac{1}{L} \left| \sum_{n=0}^{L-1} x_i[n] e^{-j2\pi fn} \right|^2$$

where f denotes the frequency and j is the imaginary unit.

4. **Averaging:** The periodograms of all segments are averaged to obtain the final Welch PSD estimate:

$$P_{Welch}(f) = \frac{1}{M} \sum_{i=1}^{M} P_i(f)$$

This averaging step effectively reduces the variance of the PSD estimate, making it more robust to noise and fluctuations in the signal.

The Welch method has several advantages compared to the traditional periodogram approach. First, by averaging the periodograms of overlapping segments, the Welch method reduces the variance of the PSD estimate. Second, the use of windowing functions minimizes spectral leakage, which leads to more accurate frequency estimates. Finally, the Welch method is computationally efficient and widely used in practical applications such as speech processing, biomedical signal analysis, and vibration analysis.

For tremor detection, the Welch method is particularly useful for analyzing the frequency content of accelerometer data collected from patients. By estimating the PSD of the accelerometer signals, it is possible to identify the dominant frequency components that are associated with tremor activity. These frequency features can then be used as input to machine learning models for classifying tremor and non-tremor cases. In this study, we implement the Welch method to compute the PSD of the windowed accelerometer signals, which can then be used as input to a classification algorithm.

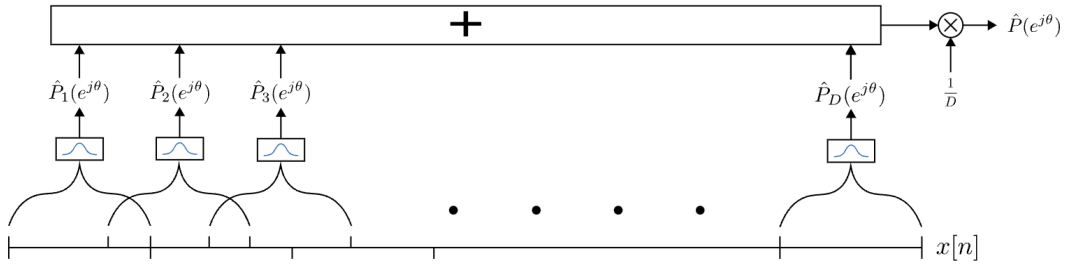

Figure 2.7: Power Spectral Density estimation using Welch's method.

2.4 Receiver Operating Characteristic Curves

Receiver Operating Characteristic (ROC) curves are a widely used tool for evaluating the performance of binary classification models. ROC curves plot the True Positive Rate (Sensitivity) against the False Positive Rate (1 - Specificity) across different classification thresholds. By visualizing these metrics, ROC curves help to understand the trade-offs between sensitivity and specificity for varying thresholds.

The ROC curve is a graphical representation that illustrates the ability of a binary classifier system to differentiate two classes as its threshold is varied. The curve is created by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) for various threshold settings.

- **True Positive Rate (TPR) or Sensitivity:** The proportion of actual positive cases that are correctly identified by the classifier. It is defined as:

$$\text{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **False Positive Rate (FPR):** The proportion of actual negative cases that are incorrectly identified as positive by the classifier. It is defined as:

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

A perfect classifier would achieve a point in the top-left corner of the ROC space, corresponding to 100% sensitivity (TPR = 1) and 0% false positive rate (FPR = 0). This point corresponds the ideal case where the classifier makes no errors.

The Area Under the Curve (AUC) is a single scalar value that summarizes the performance of a classifier across all thresholds. The AUC value ranges from 0 to 1, where:

- **AUC = 1:** The classifier perfectly distinguishes between positive and negative cases.
- **AUC = 0.5:** The classifier performs no better than random guessing.
- **AUC < 0.5:** The classifier performs worse than random guessing, which might indicate a flaw in the model or data.

In general, a higher AUC indicates better overall performance of the model.

Interpreting an ROC curve involves analyzing the trade-offs between sensitivity and specificity. Some key points to consider are:

- The closer the curve is to the upper left corner, the more accurate is the classifier.
- The closer the curve is to the diagonal line (AUC = 0.5), the less accurate is the classifier.
- The steepness of the curve is important; a steeper curve indicates a better performance.

The ROC curve can also be used to select an optimal threshold for classification by considering the point on the curve that provides the best balance between sensitivity and specificity. This optimal threshold can be found using the Youden's J statistics, which is described in the following section.

2.5 Finding the Optimal Threshold Using Youden's J Statistic

In binary classification tasks, determining the optimal threshold is crucial for maximizing the performance of the model. One effective method for identifying this threshold is by using Youden's J statistic, which balances sensitivity and specificity.

Youden's J statistic is defined as:

$$J = \text{Sensitivity} + \text{Specificity} - 1 \tag{2.5.1}$$

The value of J ranges from -1 to 1, with higher values indicating better performance of the model. The optimal threshold is the point at which J is maximized.

To determine the optimal threshold, the following steps are taken:

1. Calculate the sensitivity and specificity for each possible threshold.
2. Compute Youden's J statistic for each threshold using the formula provided.
3. Identify the threshold that maximizes Youden's J statistic.

An example of an ROC curve and the optimal threshold that is found by maximizing the Youden's J statistic is shown in Figure 2.8. For example, in this specific example optimal threshold is 0.32, maximum Youden's J statistic is 0.45, optimal FPR is 0.26, and optimal TPR is 0.71.

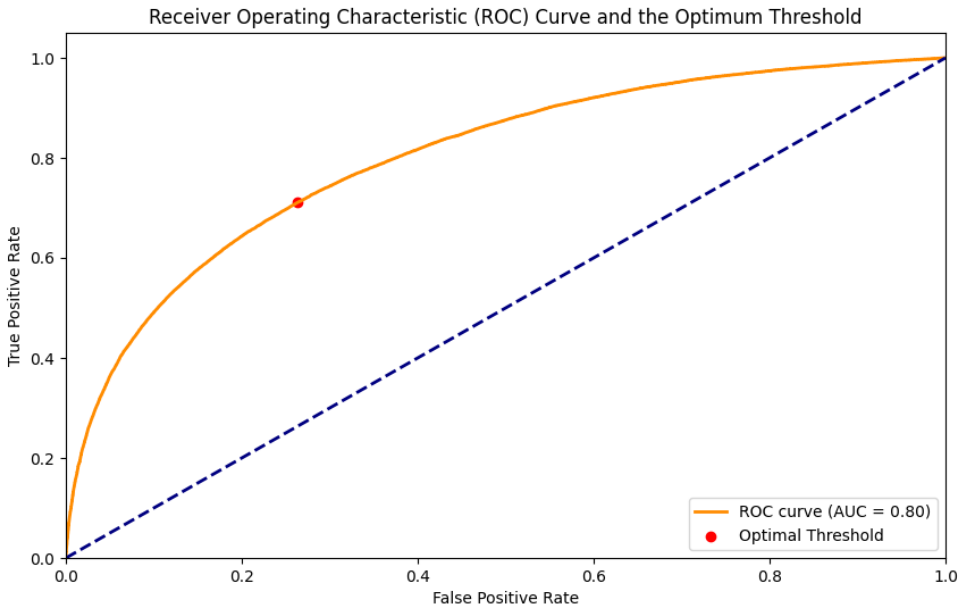

Figure 2.8: An example ROC Curve and the optimal threshold found by maximizing Youdan's J statistics.

2.6 Machine Learning Algorithms for Parkinson's Tremor Detection

Machine Learning (ML) and pattern recognition algorithms are shown to be useful in various biomedical engineering applications [21]. These algorithms essentially deal with the problem of making decisions automatically with minimal human insight [22]. Traditionally, the process is outlined using the pattern recognition pipeline as shown in Figure 2.9. In the pipeline, sensor data is first preprocessed, for example, zero-mean unit-variance normalization can be applied. Then, the relevant hand-crafted features are extracted from the preprocessed data, and these features are fed into a classification algorithm that makes predictions according to the extracted features [23]. In this more traditional approach, finding hand-crafted features can be challenging, especially when there are complex non-linear relationships between the signal and the label. To overcome this problem, more novel approaches, such as deep learning methods, can be used, which learn the features directly from the training data, without the need for hand-crafting the features [22].

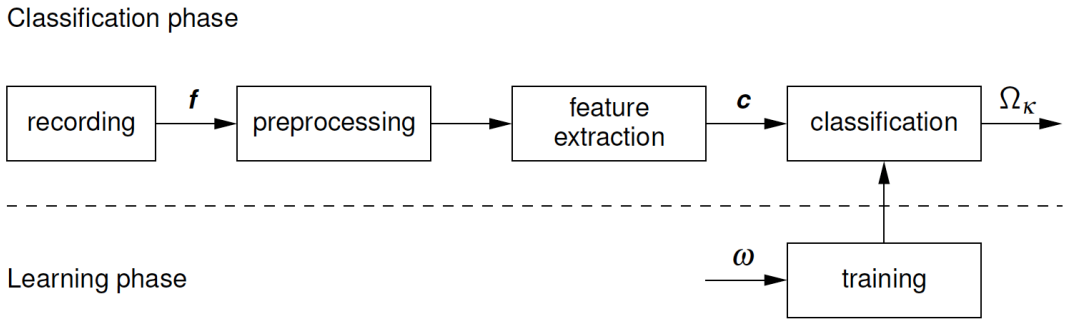

Figure 2.9: Pattern recognition pipeline [24].

Feature Extraction

To be able to solve the classification problem, we should first extract the most relevant features for classification. Suppose the feature extraction step gives us features x_1 and x_2. Then we have a feature vector $\mathbf{x}$ as,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

If we plot the feature vectors for each sample from the data in a two-dimensional feature space, each sample is represented as a point in this space. The coordinates of each point correspond to the values of the features x_1 and x_2 for that sample. For example, if we set x_1 to be the normalized energy that lies between 0-3 Hz and set x_2 to be the normalized energy that lies between the 3-9 Hz, we have a 2D feature space that looks like in the Figure 2.10. In this figure we see that the tremor cases mostly accumulate to the upper right part of the feature space, not tremor cases mostly accumulate to the lower left part of the feature space and there appears to be some overlap for the two classes. In an ideal case we should be able to choose features and get a feature space representation, where the two classes are easily seperable by the classification boundary, without any overlap of the classes. The intra-class distances for one class should be as small and inter-class distances for different classes should be as large as possible.

The features can be handcrafted as in this example or they can be learned directly by the ML model. After the boundaries that seperate different classes are learned, the model can be deployed in the real world, where the features are calculated for a given input and fed into the classifier, which classifies the given features to one of the classes.

The various ML models that are implemented in our work, both that take in hand-crafted features or that learn the feature representation from the data are explained in detail in the following sections.

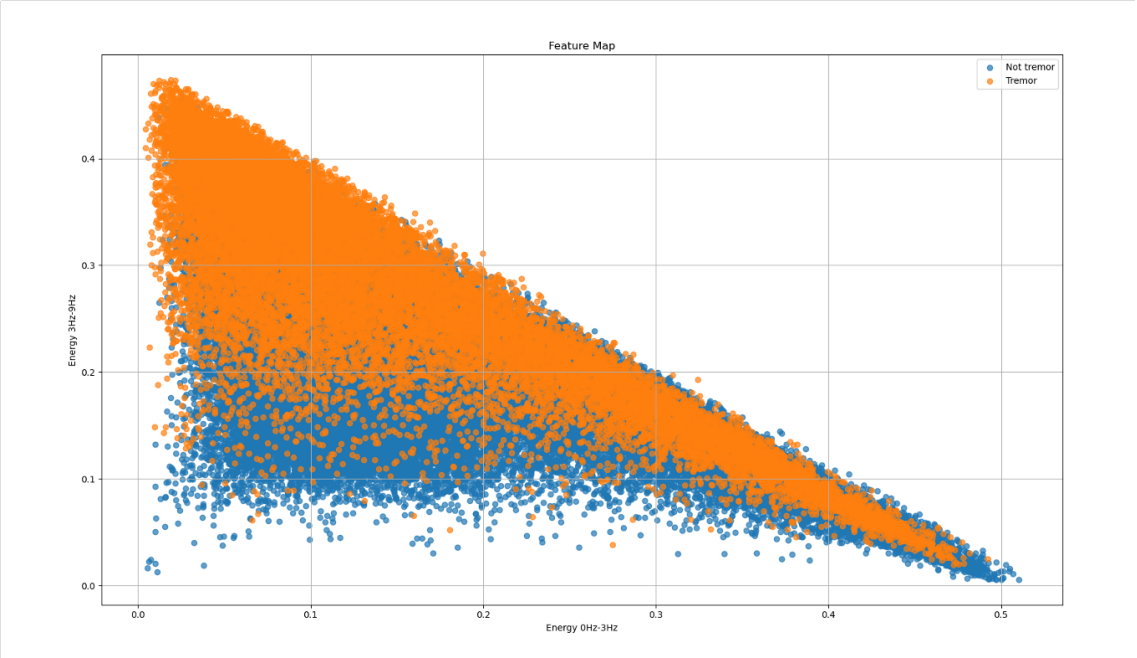

Figure 2.10: An example 2D feature space for the tremor detection problem.

Classification

Classification is a supervised learning technique used to assign input data into predefined categories or classes based on the features extracted. It involves training a model on a labeled dataset, where the correct output is known, and then using this model to predict the class labels of new, unseen data. Common classification algorithms include:

- **Linear Classifiers:** Such as Logistic Regression and Support Vector Machines (SVM), which assume a linear relationship between input features and the output class.
- **Decision Trees:** Which use a tree-like model of decisions and their possible consequences.
- **K-Nearest Neighbors (KNN):** A non-parametric method that classifies a data point based on the majority class among its nearest neighbors.

- **Neural Networks:** Complex models that mimic the structure and function of the human brain, capable of capturing non-linear relationships between features and classes.

In the following sections, we review both the more traditional ML algorithms such as support vector machines (SVM), random forest classifiers where hand-crafted features are used, and more novel deep learning approaches such as Convolutional Neural Network (CNN) where the features are learned by the neural network.

2.6.1 Support Vector Machines

Support Vector Machines (SVM) are a class of supervised learning algorithms that can be used both for classification and regression tasks. [25]. SVMs are popular classification algorithms that are used for biomedical applications such as disease classification [26].

SVMs find the hyperplane that separates data points of different classes in high dimensional space, by maximizing the margin between the hyperplane and the nearest data points, referred to as support vectors. This approach of maximizing the margin makes SVMs robust against overfitting and gives good results in generalization [25].

Suppose we have a 2-class classification task with labeled dataset as $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i is the input feature vector and y_i is the corresponding class label ($y_i \in \{-1, 1\}$), the SVM algorithm finds the optimal hyperplane by maximizing the margin between the hyperplane and the nearest data points with the following optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \tag{2.6.1}$$

subject to:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall i \tag{2.6.2}$$

where $\mathbf{w}$ is the weight vector, b is the bias term, and $\|\cdot\|$ denotes the Euclidean norm. This optimization problem can be solved using methods such as gradient descent [25].

Note that the SVM algorithm given above separates the classes linearly, however, the SVM algorithm can be extended using kernel trick so that it can also find non-linear boundaries between classes. The kernel trick maps the input feature vectors into a higher dimensional space in which the data can be separated linearly. This corresponds to a

non-linear separation in the original lower dimensional space. [25]. In this thesis, we will stay with linear SVMs.

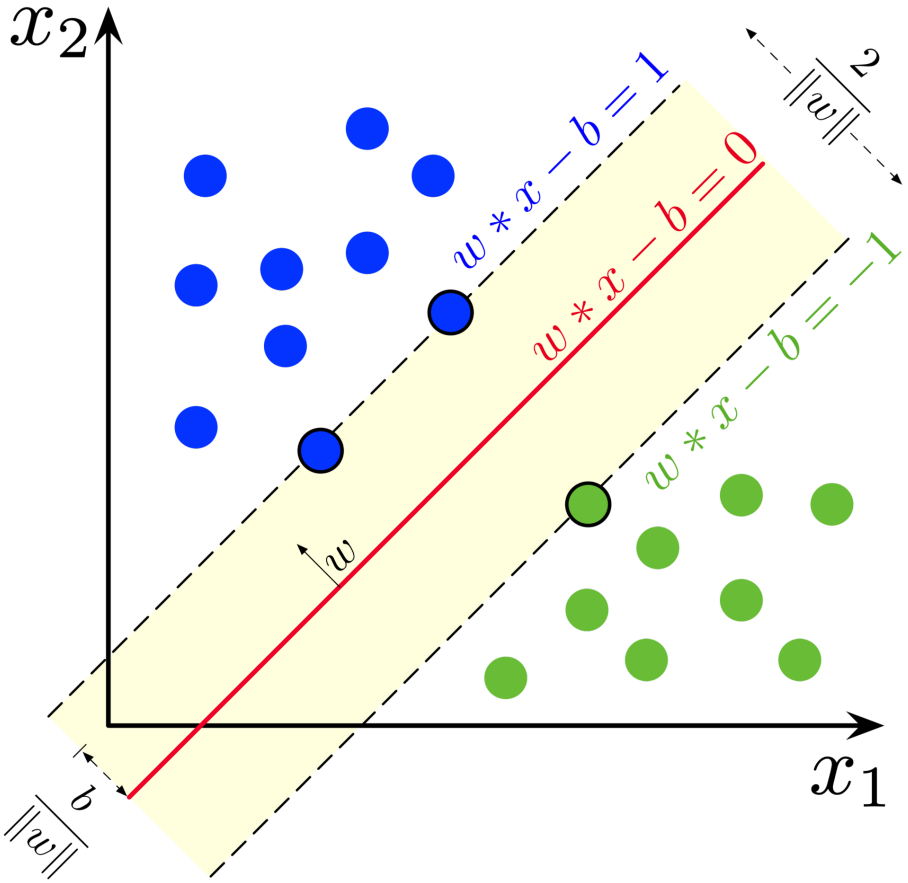

Figure 2.11: Support Vector Machine finds the optimal hyperplane that separates the two classes by maximizing the margin between the hyperplane and the nearest data points.

2.6.2 Random Forest Classifiers

Random Forest is an ensemble learning method that can be used for classification and regression tasks. The algorithm is an extension of decision tree classifiers which works by training multiple decision trees and using multiple decision trees to do classification with a voting mechanism. The combination of the predictions from multiple trees can improve the generalization performance and reduce the risk of overfitting [27]. The random forest algorithm is depicted in Figure 2.12

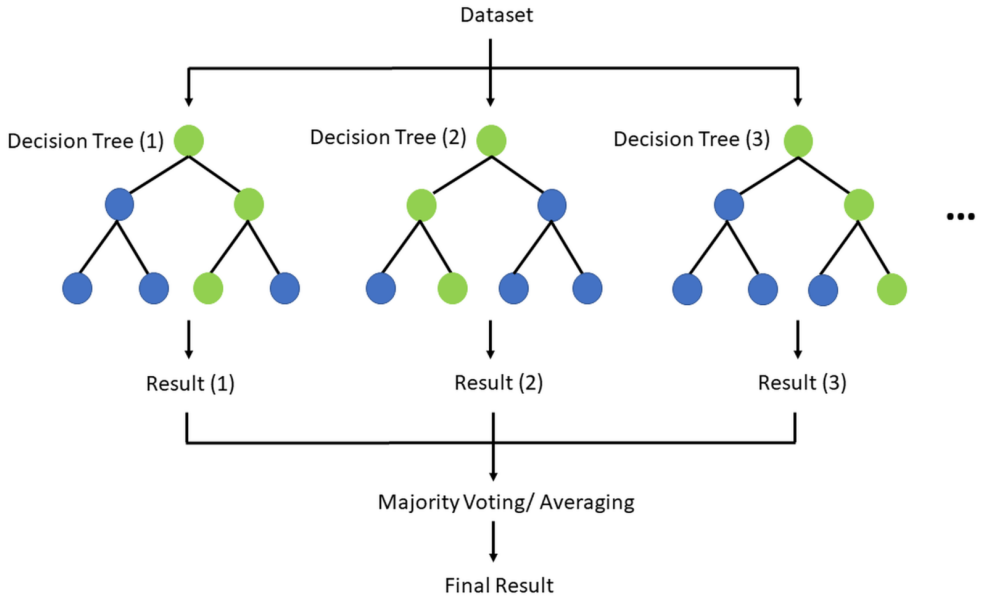

Figure 2.12: Random Forest Classifier is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes for classification tasks or the mean prediction for regression tasks [28].

2.6.3 Neural Networks

Neural networks are a subset of machine learning algorithms that are inspired by the human brain. The neural networks have layers of nodes that are interconnected to each other where each connection is associated with a weight. Neural networks can learn the features directly from the data through a backpropagation algorithm which adjusts the weights to minimize the objective function [29].

Typically a neural network consists of an input layer, hidden layers and an output layer. The input layer gets the data as the input, passes through the hidden layers, then the output layer produces the predictions. The value of each node is determined by passing the weighted sum of the connected nodes from the previous layer through a non-linear activation function. This structure allows the neural network to learn the features directly from the data by capturing non-linear relations in the data, making neural networks powerful machine learning methods [30].

Table 2.1: Common activation functions used in neural networks

Name	Mathematical Description
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$
Hyperbolic Tangent (Tanh)	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
Rectified Linear Unit (ReLU)	$\text{ReLU}(x) = \max(0, x)$
Leaky ReLU	$\text{Leaky ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$
Parametric ReLU (PReLU)	$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$
Exponential Linear Unit (ELU)	$\text{ELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha(e^{x}-1), & \text{if } x < 0 \end{cases}$
Softmax	$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
Swish	$\text{Swish}(x) = \frac{x}{1+e^{-x}}$
GELU (Gaussian Error Linear Unit)	$\text{GELU}(x) = 0.5x\left(1+\tanh\left(\sqrt{\frac{2}{\pi}}\left(x+0.044715x^{3}\right)\right)\right)$

Fully Connected Network

Fully Connected Network (FCN) is one of the most basic neural network architectures. FCN consists of layers, where each node in the layer is connected to every node in the next layer with associated weights. The output vector is found by multiplying the input vector $\mathbf{x}$ with the weight matrix $\mathbf{W}$, adding the bias vector to it, and passing the result through a non-linear activation function f as described in Equation 2.6.3

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.6.3}$$

where $\mathbf{x}$ is the input vector, $\mathbf{W}$ is the weight matrix, $\mathbf{b}$ is the bias vector, f is the activation function, and $\mathbf{y}$ is the output vector of the layer.

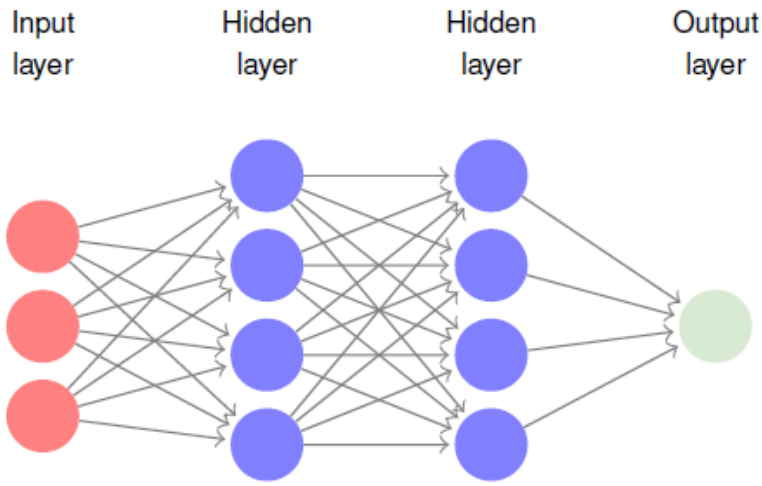

Figure 2.13: A fully connected network with three input nodes, one output node and two hidden layers, each with 4 nodes. [31]

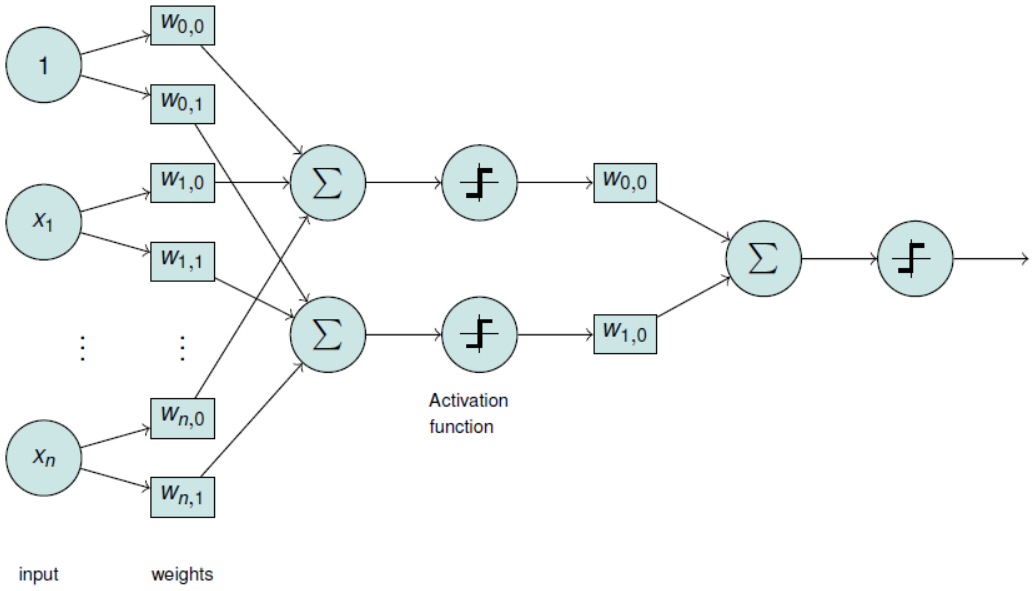

Figure 2.14: Basic architecture of a feedforward neural network. Inputs $x_1, \ldots, x_n$ are weighted, summed, and passed through activation functions, producing the output. [31]

Convolutional Neural Network

Even though the FCNs are powerful neural networks that can find non-linear patterns in the input data, the main limitation of this method is that it does not consider spatial dependencies between the samples. This is a major disadvantage when using FCNs with time series data, where an input sample from one time instant is dependent on the samples from the adjacent time instants, or when using FCNs on image data, where neighboring pixels are dependent on each other. CNN is a specialized type of neural network that can take into account the dependencies between the input samples.

CNNs are usually composed of convolutional layers that is mathematically described as in Equation 2.6.4, where convolutional kernels are convolved across the input data to produce feature maps. Each filter can detect specific patterns that are learned from the training data [32].

$$\mathbf{y}_{i,j}^{k} = f\left(\sum_{m,n} \mathbf{W}_{m,n}^{k} \mathbf{x}_{i+m,j+n} + \mathbf{b}^{k}\right) \tag{2.6.4}$$

where $\mathbf{y}_{i,j}^{k}$ is the output feature map, $\mathbf{W}_{m,n}^{k}$ is the k-th filter, $\mathbf{x}_{i+m,j+n}$ is the input patch, $\mathbf{b}^{k}$ is the bias term, and f is the activation function, typically ReLU [32].

Each convolutional layer is usually followed by a pooling layer, which reduces the dimensions of the feature maps to reduce the computational load and adds additional non-linearity. Various types of pooling operations exist, however the most common type is the max pooling, which takes the maximum value from a region of the feature map, mathematically described as,

$$\mathbf{y}_{i,j}^{k} = \max(\mathbf{x}_{i+m,j+n}) \tag{2.6.5}$$

After the input data is passed through a couple of convolutional layers that are followed by pooling layers, the output of the last pooling layer is flattened and passed through an FCN to obtain the final prediction of the network. This part of the neural network works exactly as described above in the section Fully Connected Layers (FCNs).

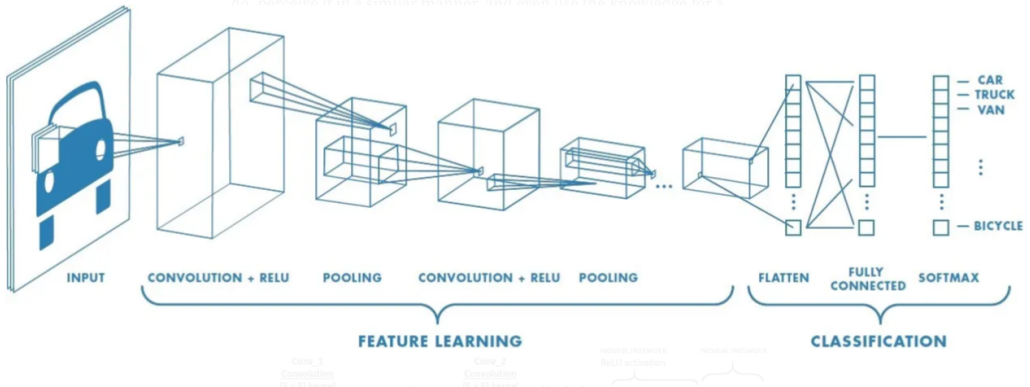

Figure 2.15: Basic architecture of a convolutional neural network. Convolutional layers, together with the pooling layers extract the features from the input. Then the fully connected layers do the classification. [33]

Recurrent Neural Networks

Recurrent Neural Network (RNN) is a class of neural networks that are specially designed to recognize patterns in the sequences of data, such as time series, speech, and text.

The main idea of RNNs is that they maintain a hidden state that is adjusted using the previous input sample and which affects the next output sample, thereby modeling the temporal dependencies between the adjacent input samples. The main building block of an RNN is called a recurrent cell, which processes input samples one by one while maintaining and updating the hidden state, where the hidden state at time step t is computed as given in Equation 2.6.6

$$\mathbf{h}_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h) \tag{2.6.6}$$

where $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ are the current and previous time steps $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$ are weight matrices, $\mathbf{b}_h$ is the bias vector, and f is the activation function, typically a hyperbolic tangent (tanh) or ReLU.

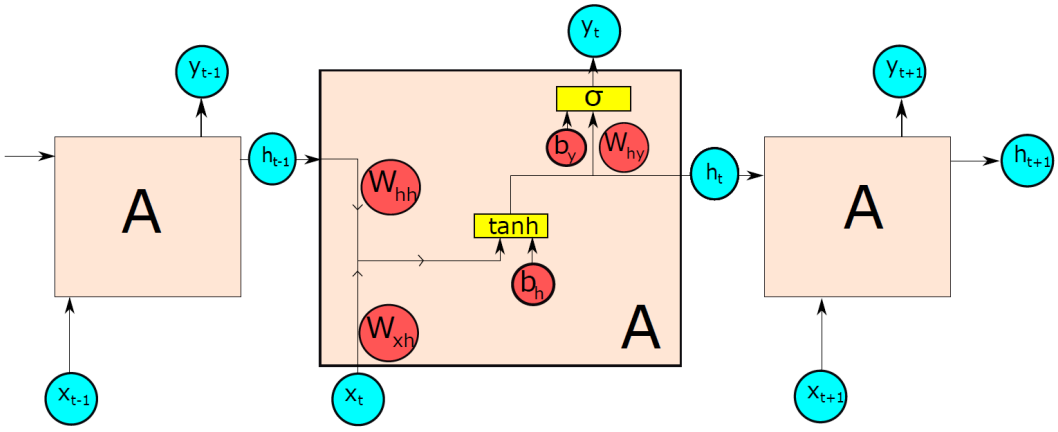

Figure 2.16: Unfolded architecture of the basic recurrent neural network. The output y_t is computed from the updated hidden state h_t using weights W_{hy} and biases b_y, passed through a σ activation function. Hidden state h_t is transferred to the next time step, enabling the sequence modelling. [31]

2.7 Optimization and Regularization

2.7.1 Optimization of Neural Networks

For neural networks to be able to learn from data and generalize well to unseen data, we have to do optimization during the training of the neural network. The goal of the optimization is to minimize a loss function, which is a measure of how much discrepancy is there between the predicted outputs and true labels.

Gradient Descent

The most common method used to optimize neural networks is called *gradient descent*. Gradient descent iteratively adjusts the network parameters (weights and biases) to reduce the loss function. The parameters θ are updated as follows:

$$\theta := \theta - \eta \nabla_\theta J(\theta), \tag{2.7.1}$$

where η is the learning rate, and $\nabla_\theta J(\theta)$ represents the gradient of the loss function $J(\theta)$ with respect to the parameters θ.

There exists some variants of gradient descent. The most common ones are:

- **Batch Gradient Descent:** The parameters are updated using the entire dataset, which makes it computationally very expensive for large datasets.
- **Stochastic Gradient Descent (SGD):** The parameters are updated for each data point in the dataset. Computationally easier and faster but introduces noise into the computation of gradients
- **Mini-batch Gradient Descent:** The parameters are updated using small batches of data. This method is a compromise between the batch and stochastic gradient descent methods, to find a good trade-off between computational efficiency and stability.

The choice of the learning rate η is very critical in gradient descent. Too high learning rates may cause divergence by overshooting of the minima of the loss function, whereas, too low learning rates may result in too slow convergence. There exist some methods to adaptively set the learning rate.

- **Momentum:** Momentum accelerates gradient descent by adding a fraction of the previous update to the current update:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta J(\theta) \tag{2.7.2}$$

  where v_t is the velocity term, β is the momentum coefficient, and v_{t-1} is the previous velocity.

- **Adam (Adaptive Moment Estimation):** Adam combines the momentum and adaptive learning rates by using the estimates of first and second moments of gradients:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta) \tag{2.7.3}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla_\theta^2 J(\theta) \tag{2.7.4}$$
$$\theta = \theta - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.7.5}$$

Here, m_t is the first moment (mean) and v_t is the second moment (uncentered variance), β_1 and β_2 are decay rates, and ϵ is a small constant for numerical stability.

2.7.2 Regularization in Neural Networks

Regularization is a critical concept in machine learning, particularly in deep learning, where models often have a large number of parameters. The primary goal of regularization is to prevent overfitting, where a model performs well on training data but fails to generalize to unseen data. Overfitting occurs when a model becomes too complex, capturing not only the underlying patterns but also the noise in the training data. Regularization techniques mitigate overfitting by imposing constraints or incorporating additional information, which leads to simpler, and more generalizable models. Example cases for underfitting, appropriate fitting and overfitting are given in Figure 2.17.

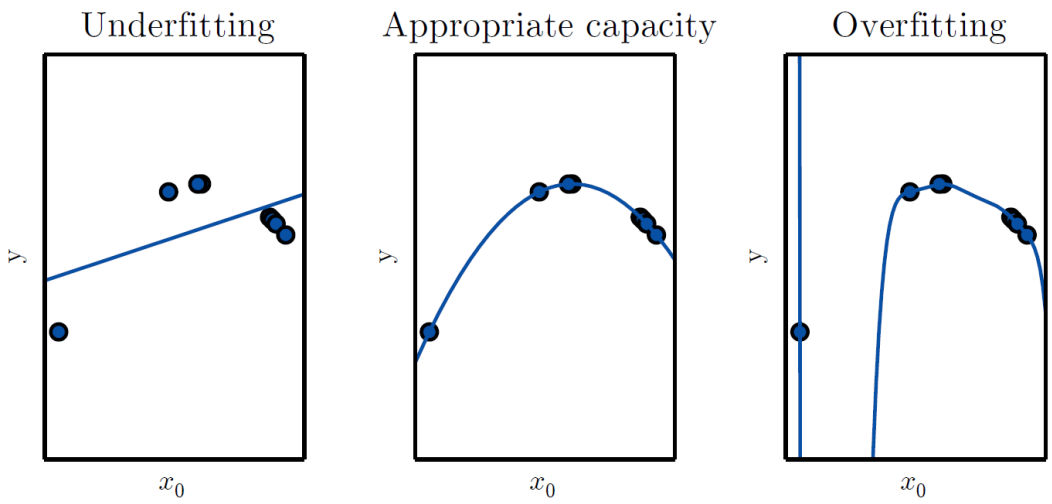

Figure 2.17: Three models to model the distribution of the data. We want a model to have appropriate capacity, as in the middle, so that it can generalize well to unseen data. [30]

L2 Regularization

L2 regularization, also known as Ridge Regression, is one of the most commonly used regularization techniques. It works by adding a penalty term to the loss function, propor-

tional to the sum of the squared values of the model's weights. The loss function with L2 regularization is defined as:

$$\mathcal{L}_{\text{L2}} = \mathcal{L} + \lambda \sum_{i=1}^{n} w_i^2$$

Where:

- $\mathcal{L}$ is the original loss function.
- w_i are the model's weights.
- λ is the regularization parameter that controls the strength of the penalty.

L2 regularization discourages large weights by shrinking them, which helps to prevent overfitting. It encourages the model to distribute the importance across all features rather than relying heavily on a few. This is particularly important in deep learning models with many parameters, as it promotes generalization and reduces sensitivity to noise in the training data.

L2 regularization is implicitly applied through the use of the Adam optimizer, which includes a weight decay term. The weight decay in Adam serves the same purpose as L2 regularization, penalizing large weights and helping the model generalize better.

Dropout Regularization

Dropout is a regularization technique specifically designed for neural networks. Proposed by [34], dropout involves randomly setting a fraction of the neurons in a layer to zero during each training iteration. This prevents the network from becoming overly reliant on any single neuron, thereby promoting the development of robust, distributed representations.

During the training phase, each neuron's output is retained with a probability p and set to zero with a probability $1 - p$. The modified forward pass for a layer with dropout can be expressed as:

$$\text{output} = \frac{1}{p} \cdot \text{mask} \cdot \text{activation}$$

Where:

- *mask* is a binary vector that determines which neurons are dropped.
- p is the probability of retaining a neuron.

During testing, dropout is not applied, and the full network is used, but the outputs are scaled by p to account for the neurons that were dropped during training.

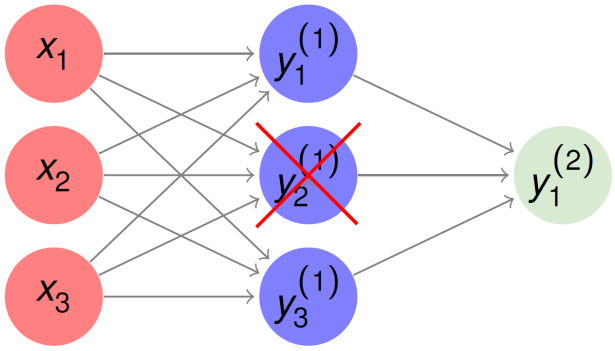

Figure 2.18: Dropout regularization randomly sets some activations to zero with probabilty $1-p$. [31]

Data Augmentation

Data augmentation is a regularization technique that involves creating new training samples by applying various transformations to the original data. This is particularly useful in scenarios where the amount of training data is limited, as it allows the model to learn from a more diverse set of examples, thereby improving its ability to generalize.

For time-series data, such as the accelerometer readings used in Parkinson's tremor detection, augmentation might involve adding noise, scaling, or jittering the data.

Early Stopping

Early stopping is a regularization technique that monitors the model's performance on a validation set during training. If the model's performance on the validation set stops improving and starts to degrade, indicating overfitting, the training process is halted.

The model parameters are then reverted to the state corresponding to the best validation performance.

In practice, early stopping involves setting a patience parameter, which defines the number of epochs to wait for an improvement before stopping the training. If the validation loss does not improve for a specified number of epochs, training is stopped, and the best model weights are restored.

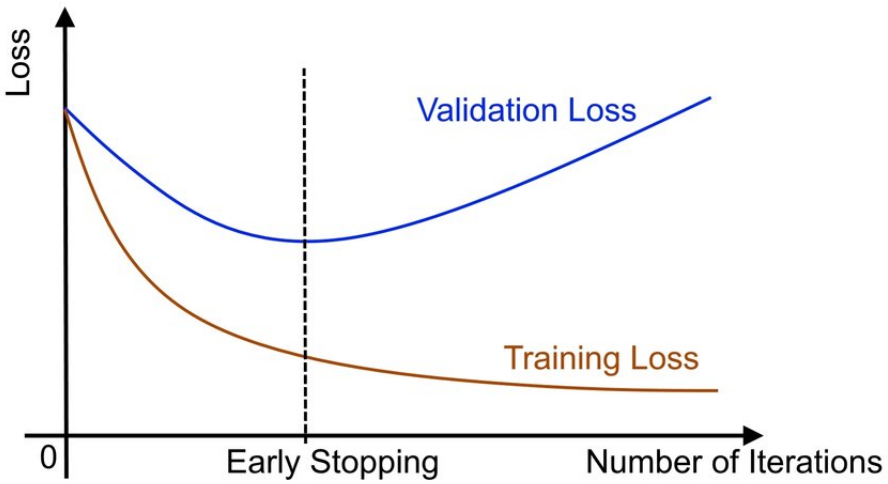

Figure 2.19: Early stopping terminates the training when the performance on the validation set does not improve, to avoid overfitting.

Combining Regularization Techniques

In practice, multiple regularization techniques are often combined to achieve the best performance. For example, dropout, L2 regularization, data augmentation and early stopping can be used together to ensure that the model generalizes well across various conditions. Each technique addresses overfitting from a different perspective, and their combined effect can lead to a more robust model.

2.8 Quantization and Pruning to Implement Neural Networks on Resource Constrained Environments

2.8.1 Weight Quantization

The machine learning algorithms described above will be implemented on an embedded device, enabling Parkinson's tremor detection to be performed in real time. Embedded devices, which include microcontrollers, smartphones, and IoT (Internet of Things) devices, are specially designed for specific tasks, often characterized by limited resources such as memory, computing power, and storage capacity. These constraints necessitate a different approach to deploying machine learning models compared to typical computing environments.

One of the key differences between embedded devices and general-purpose computers is the word size. Embedded devices commonly operate on 8-bit integers, while typical computers generally handle 32 or 64-bit floating-point values. This discrepancy in word size means that the parameters of a machine learning model, which are often represented as 32-bit or 64-bit floating-point numbers during training, must be reduced or quantized before deployment on an embedded device. Quantization is a critical step that involves mapping the parameters of a model to a lower precision representation, thereby reducing the model's size and computational requirements. However, this reduction in precision can lead to a trade-off between model performance and resource efficiency.

There are several quantization methods available, each with varying impacts on the performance of the deployed model [35]. These methods include:

- **Post-Training Quantization:** Post-training quantization involves converting a trained model to a lower precision format after the training process is complete. This method does not require any changes to the training procedure and is relatively simple to apply. It includes techniques such as full integer quantization, where weights and activations are both quantized to integers, and mixed precision quantization, where different parts of the model are quantized to different precisions based on their sensitivity to precision loss.
- **Quantization-aware training:** Quantization-Aware Training (QAT) is a technique where the model is trained with quantization in mind. During training, fake quantization is applied to weights and activations, simulating the effects of quantization

on the model's performance. This allows the model to learn to adapt to the lower precision, often resulting in a model that is more robust to the precision loss that occurs during quantization. QAT typically yields better accuracy than post-training quantization but requires more computational resources during training.

Quantization, while beneficial for deploying models on resource-constrained devices, can introduce some challenges. The primary challenge is the potential degradation in model accuracy due to the reduced precision. Careful selection of the quantization method and extensive testing are required to ensure that the model performs adequately in the deployment environment. Moreover, specific hardware capabilities of the embedded device, such as support for certain quantization levels, need to be considered during the model conversion process.

In conclusion, quantization is an essential process that enables the deployment of complex machine-learning models on embedded devices. By carefully choosing the appropriate quantization technique and understanding the trade-offs involved, it is possible to achieve a balance between model performance and resource efficiency, facilitating the real-time detection of Parkinson's tremor on embedded devices.

2.8.2 Neural Network Pruning

Neural network pruning is a technique used to reduce the size of a neural network by removing less important weights, neurons, or layers, thus making the model more efficient without significantly compromising its performance. The primary goal of pruning is to reduce the computational complexity, memory usage, and inference time, which are crucial for deploying deep learning models in resource-constrained environments, such as embedded devices.

Pruning techniques can be broadly categorized into three types: *unstructured pruning*, *structured pruning*, and *dynamic pruning* [36].

- **Unstructured Pruning**: This approach involves the removal of individual weights from the neural network based on a certain criterion, such as the magnitude of the weights. Weights with small magnitudes contribute less to the overall model output and can be removed with minimal impact on the network's accuracy. Although unstructured pruning can significantly reduce the number of parameters, it often

leads to sparse weight matrices, which may require specialized hardware or libraries to achieve computational speedup.

- **Structured Pruning**: In structured pruning, entire neurons, filters, or even layers are removed from the network. This approach results in a more compact and efficient architecture that can be directly deployed on existing hardware without the need for sparse matrix operations. Common structured pruning techniques include pruning channels in convolutional layers or pruning entire layers based on their contribution to the network's output.
- **Dynamic Pruning**: Unlike static pruning, which is performed before or after training, dynamic pruning involves adjusting the network's architecture during training. This technique can adaptively add or remove weights, neurons, or layers in response to changes in the loss function or other metrics. Dynamic pruning aims to find an optimal balance between model size and performance throughout the training process.

We can illustrate the unstructured and structured pruning in Figure 2.20, referring to how the pruning is performed in a weight matrix of a model. [37]

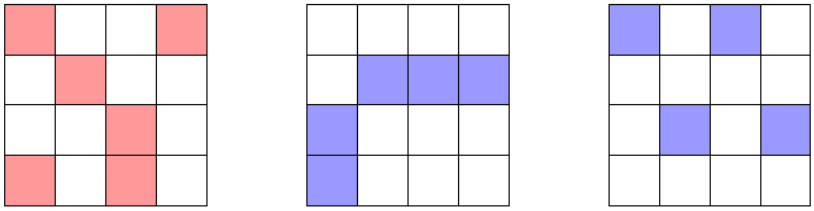

Figure 2.20: Unstructured pruning (left panel) versus structured pruning (middle and right panels). [37]

Several methods have been proposed for neural network pruning, each with its own set of advantages and limitations [36]:

- **Magnitude-Based Pruning**: This method removes weights with the smallest magnitudes, under the assumption that such weights have the least impact on the network's output. It is simple to implement and can be applied to both fully connected and convolutional layers.

- **Sensitivity-Based Pruning**: Sensitivity-based pruning measures the change in the loss function when a weight or neuron is removed. Weights or neurons with a minimal impact on the loss function are pruned. This approach can achieve a higher degree of compression but may involve more computational overhead compared to magnitude-based pruning.
- **Regularization-Based Pruning**: Incorporating regularization techniques, such as L1 or L2 regularization, during training can encourage sparsity in the weights. By adding a penalty term to the loss function, the network learns to reduce the magnitude of certain weights, making them candidates for pruning.
- **Lottery Ticket Hypothesis**: This approach is based on the idea that there exists a smaller subnetwork, referred to as the winning ticket which, when trained in isolation, can achieve comparable performance to the original larger network. Identifying this subnetwork through iterative pruning and retraining can yield a more efficient model.

While pruning can effectively reduce the number of parameters and computational complexity, it can also potentially degrade the model's performance if not done carefully. To mitigate this, fine-tuning is often performed after pruning to restore the model's accuracy. In some cases, iterative pruning and fine-tuning cycles can be used to achieve higher levels of compression without significant loss of accuracy.

Neural network pruning has been widely used in various applications, such as deploying deep learning models on edge devices, accelerating inference in real-time applications, and reducing the energy consumption of neural networks. By optimizing the model's size, pruning allows for faster computation, lower memory requirements, and increased power efficiency, making it a crucial technique for practical deep learning deployment.

Despite its benefits, neural network pruning poses several challenges. Determining the optimal amount of pruning without compromising the model's generalization capabilities remains an open problem. The effect of the amount of unstructured and structured pruning on the model performance is investigated in detail in this study.

2.9 Edge Artificial Intelligence and Tiny Machine Learning

Edge Artificial Intelligence (Edge AI) refers to the deployment of AI algorithms directly on edge devices, such as smartphones, IoT devices, and other embedded systems [38]. Edge AI allows for data processing to occur at the source of data generation, rather than relying on centralized cloud servers. This decentralized approach offers several advantages, including reduced latency, enhanced privacy, and decreased dependency on constant network connectivity [39].

The shift towards Edge AI is driven by the increasing demand for real-time AI applications across various domains, including healthcare, automotive, and industrial automation. In the context of Parkinson's tremor detection, Edge AI enables the deployment of machine learning models on portable or wearable devices, such as smartwatches or medical sensors, allowing for continuous monitoring and analysis of patient data without the need for constant cloud connectivity.

To effectively implement Edge AI, several considerations must be addressed. These include optimizing the model for the limited computational resources available on edge devices, ensuring efficient energy consumption, and managing the trade-offs between model accuracy and resource usage. Techniques such as model compression, quantization, and pruning (as discussed in the previous sections) are commonly employed to enable AI models for edge deployment.

Furthermore, advances in hardware design have played a pivotal role in the rise of Edge AI. Modern edge devices are equipped with specialized hardware accelerators, such as GPUs, TPUs, and dedicated AI processors, which are optimized for the efficient inference with AI. These advancements enable complex ML models to be deployed on edge devices [40].

In summary, Edge AI enables ML models to run on embedded devices. For Parkinson's tremor detection, this means enabling real-time analysis and monitoring on portable or wearable devices, and therofore improving patient care and and their quality of life.

Tiny Machine Learning (TinyML) refers to the deployment of machine learning algorithms on low-power microcontroller units (MCUs) and other embedded devices with constrained computational resources. TinyML represents a convergence of embedded systems, machine learning, and IoT (Internet of Things), enabling intelligent data processing directly at the edge. Unlike traditional ML workflows that involve sending data to powerful cloud servers

for inference, TinyML brings the capabilities of machine learning closer to the source of data collection, thus enabling real-time processing, reducing latency, and ensuring data privacy [37].

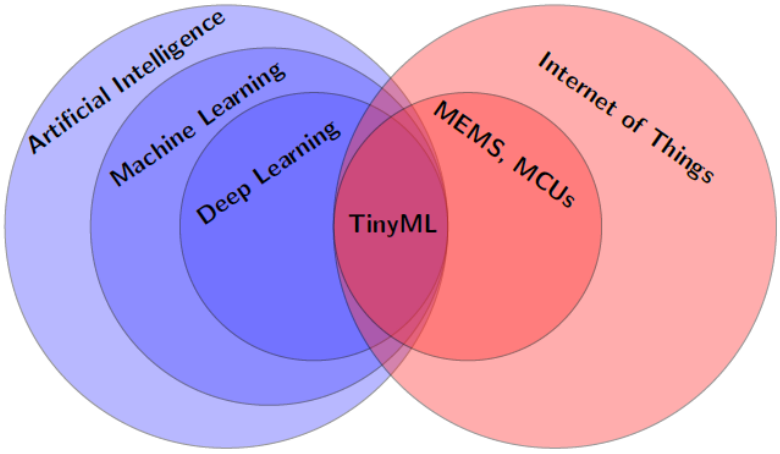

Figure 2.21: TinyML is at the intersection of artificial intelligence and internet of things (IoT). [37]

The motivation for TinyML arises from the limitations of cloud-based machine learning and the growing demand for low-latency, energy-efficient, and privacy-preserving solutions. In many applications such as wearable health monitoring, environmental sensing, and smart home automation, continuous data transmission to the cloud can be impractical due to bandwidth constraints, energy consumption, and privacy concerns. TinyML addresses these challenges by allowing devices to perform local inference, making it feasible to deploy intelligent algorithms in resource-constrained environments.

However, the adoption of TinyML comes with significant challenges [37]:

- **Computational Constraints:** Typical microcontrollers, with clock speeds ranging from 1-100 MHz and memory sizes in the kilobytes to a few megabytes, provide a fraction of the computational power available in modern CPUs or GPUs.
- **Power Efficiency:** Many applications require devices to run on batteries or energy-harvesting solutions, making energy-efficient algorithms critical for prolonged device operation.

- **Limited Storage and Memory:** TinyML models must be compact and optimized to fit within the small memory footprints available on embedded devices.
- **Real-Time Performance Requirements:** Many edge applications require rapid inference times to support real-time decision-making.

Addressing the challenges of deploying machine learning on microcontrollers involves a variety of techniques that optimize both the models and the hardware [37]:

- **Model Compression:** Techniques such as quantization, pruning, and knowledge distillation are employed to reduce model size and complexity. Quantization involves converting floating-point weights to lower precision (e.g., 8-bit integers), significantly reducing memory usage and computational overhead. Pruning eliminates less important connections in the network, while knowledge distillation transfers the knowledge from a large model (teacher) to a smaller model (student).
- **Efficient Neural Network Architectures:** Lightweight models such as MobileNet, SqueezeNet, and Tiny-YOLO are designed specifically for low-resource environments. These architectures often use depthwise separable convolutions or grouped convolutions to reduce the number of parameters and computation requirements.
- **Hardware Acceleration:** Specialized hardware accelerators (e.g., ARM's Cortex-M series with DSP extensions, and Google's Edge TPU) can significantly speed up model inference while maintaining low power consumption. These accelerators are designed to execute operations like matrix multiplication and convolution more efficiently than general-purpose MCUs.
- **On-device Training:** Although most TinyML applications focus on inference, on-device training is becoming an area of research interest. Techniques such as federated learning enable distributed model training across multiple devices, preserving privacy by keeping data local.

TinyML is becoming popular especially in healthcare industry [41]. Wearable devices with embedded sensors can utilize TinyML for health monitoring tasks, such as detecting abnormal heart rhythms, classifying sleep stages, or predicting seizures. Since inference happens on the device, privacy is enhanced, and real-time alerts can be generated without cloud connectivity.

To conclude, TinyML represents a transformative shift in AI, as it makes it possible to deploy machine learning models on resource constraint environments, such as embedded

devices. As the capabilities of embedded devices and optimization techniques continue to advance, the range of potential applications will expand, enabling innovation across various fields, particularly in healthcare.

Having established the theoretical background, we now turn to a review of recent literature to understand the current state of Parkinson's tremor detection. Then, we review the current literature on Edge AI and TinyML.

3 Literature Review on Parkinson's Tremor Detection and Edge AI

This section presents an extensive review of prior work in Parkinson's tremor detection, examining both traditional and novel ML methods. This section also presents the literature on Edge AI and TinyML.

Wearable devices and IMU sensors are used in the field of Human Activity Recognition (HAR), with applications in healthcare, military, and consumer contexts. Lara et al. [42] makes a extensive review on HAR systems and the algorithms that are mainly used. The paper reviews feature extraction techniques, including time-domain and frequency-domain analysis, which transform raw data into meaningful features. The paper also reviews machine learning classifiers like decision trees, neural networks, and ensemble methods commonly used in HAR systems. The paper also discusses the potential of wearable-based HAR for real-time applications, where limitations in accuracy, energy consumption, and user flexibility remain open areas for future research.

There have been numerous studies focused on using wearable devices for tremor detection. Specifically, IMU data is used in these studies, and IMU data is shown to be useful in tremor detection both inside and outside of the clinical environment.

De et al. conduct an extensive review which discusses the current applications of machine learning in tremor analysis, with a particular focus on its potentials and limitations [43]. They show that machine learning applied to accelerometric data gives promising results in the detection of Parkinson's tremor. The authors discuss common challanges in this area, such as inconsistent diagnostic criteria, non-standardized experimental setups, and small sample sizes, which limit generalizability and clinical application. They show that for effective classification of Parkinson's tremor, linear models like SVMs and ensemble algorithms, as well as deep learning models like CNN and RNN are used.

Pasluosta et al. discuss the possibility of monitoring PD patients outside the lab environment by leveraging a laterally distributed platform, such as the Internet of Things (IoT) [5]. They argue that real-time motion metrics could be obtained for PD patients by placing lightweight sensors on the patient's clothes and obtaining data from these sensors, which

could be directed to a medical database and further processed using intelligent algorithms. By using such a technology, it is possible to gather huge amounts of patient data outside the clinical environment. The collected data could be used to detect threatening conditions, adjust medications accordingly, and plan strategies to modify disease progression.

Barth et al. make use of IMU sensors, apply gait analysis and measure gait patterns in PD to distinguish mild and severe impairment of gait [44]. They extract 12 features from IMU sensors placed on the shoe and use these features in a classifier to detect PD symptoms. The method they use is successful in identifying PD-associated gait patterns and distinguishing different levels of gait impairments.

Eskofier et al. discuss the use of wearable sensor technology to monitor the motor symptoms in PD patients with a particular focus on using deep learning to detect bradykinesia, a major symptom other than tremor [45]. The motivation is similarly to accurately monitor the bradykinesia to enhance the treatment and therapy of PD patients. To detect bradykinesia, they use IMU data, which they collect through specific tasks such as finger-to-nose movements and pronation-supination exercises. In this study, they compare various traditional machine learning methods, such as SVM, kNN and AdaBoost, with deep learning approaches, particularly convolutional neural networks (CNNs). They conclude that the deep learning approaches, which can automatically extract features from raw data, outperform traditional machine learning techniques in detecting bradykinesia, with an accuracy improvement of at least 4.6%.

Sigcha et al. propose a system for the automatic detection of resting tremors in PD patients using smartwatches equipped with triaxial accelerometers [46]. They gathered accelerometer data from 18 PD patients performing some activities while wearing a smartwatch. The system they developed collects the motion data and assesses tremors based on their severity. To accomplish this they use a CNN to classify both the context (whether the patient is at rest) and the presence of tremors simultaneously. Their system achieved high accuracy in detecting tremors, with performance closely aligning with clinical assessments.

San-Segundo et al. present a study using wrist-worn accelerometers to detect tremors in PD patients in both laboratory and real-world settings [47]. The study discusses various feature sets and ML algorithms for detecting PD tremors using IMU data, also considering real-world challenges such as weak labeling, high patient variability, and the distinction between tremors and daily activities. They gathered data from PD patients in both controlled laboratory settings and real-world scenarios. Then, they compared several

feature sets, such as energy thresholds in the 3-9 Hz band, traditional statistical features (e.g., mean, variance), and learned features through CNNs. They also focused on detecting tremors in the wild (in the real world), which is a challenging problem due to the weak labeling of the data. They employed weakly supervised learning algorithms that could work with imprecise labeling of the data. The method they propose integrates a tremor spectrum extraction technique into a CNN that is called CNN-T/NT. This method is shown to be the best-performing model in tremor detection, which outperforms traditional ML algorithms such as Random Forests and Support Vector Machines.

Papadopoulos et al. presents a novel approach for detecting Parkinson's tremor using IMU data gathered in real-world, unsupervised setting [48]. They propose multiple-instance learning, where each subject is represented by a bag of IMU signal segments and labeled based on whether tremor was observed. This approach can handle coarse labels without requiring continuous annotations. The model they use employs an attention-based pooling layer that identifies critical instances where segments likely containing tremor within the bag, which improves classification performance and interpretability. They use a dataset which includes IMU data from 45 subjects, both PD patients and healthy controls, gathered through a mobile app. This in-the-wild approach introduces realistic noise and variability to the noise compared to controlled lab environments. The proposed method provides a promising solution for PD tremor monitoring outside the lab environment.

The thesis written by Zhang explores methods for detecting Parkinson's disease (PD) symptoms, particularly tremors, using wearable accelerometers and machine learning models. The thesis emphazises two main methods [49]. First, it proposes a technique named stratified weakly supervised learning, which is a technique for training on stratified labels, which approximate the percentage of time tremor symptoms occur. Second, it examines personalization algorithms such as kernel mean matching and selective transfer machine, that adjusts models to individual patients' data without requiring labeled data for each new user. The author argues that the personalized algorithms improve symptom detection compared to generic classifiers.

Apart from the increasing interest in using machine learning algorithms on tremor detection, there has been an increasing interest in deploying artificial intelligence on edge devices, namely Edge AI. Singh et al. provide a comprehensive overview of Edge AI, focusing on the integration of AI with edge computing [39]. The survey highlights the growing importance of deploying AI algorithms on edge devices to process data locally, reducing latency and bandwidth usage. The authors focus on various applications of Edge AI,

including real-time decision-making in healthcare systems. Furthermore, they discuss the challenges associated with Edge AI, such as limited computational resources and energy efficiency on edge devices. The paper emphasizes the role of Edge AI in enabling intelligent, low-latency processing directly on the device, which is critical for time-sensitive applications.

Roth et al. provide an overview of resource-efficient neural network techniques, with particular focus on embedded applications, where the available computational and energy resources are limited [50]. The methods that are mainly discussed in the paper are quantization, network pruning, and structural efficiency. They discuss that quantization decreases the memory usage and computational load by reducing the number of bits to repreresent weights and activations, and thereby achieving fast, low-energy inference on embedded hardware. Then, network pruning is discussed, which can reduce the size of the models significantly, by pruning either individual weights or whole structures (such as channels or neurons), while maintaining performance. Moreover, structural efficiency is discussed, which is about implementing efficient designs, such as knowledge distillation (where a small model learns from a larger, complex model), weight sharing, and lightweight architecture designs, to reduce complexity without a major impact on accuracy. The paper also discusses various hardware options for deploying these resource-efficient networks, including CPUs, GPUs, FPGAs, and specialized accelerators, and presents experiments comparing different compression methods on standard datasets. The work emphasizes the challenge of balancing efficiency and prediction quality, especially in constrained environments like embedded devices.

Le et al. makes a comprehensive review on Tiny Machine Learning (TinyML), with a particular focus on efficient neural networks for deployment on low-power microcontrollers (MCUs) [37]. They argue that TinyML enables AI-powered applications on resource-limited devices by using model optimization techniques, such as model compression, quantization, and pruning, to reduce computational and memory demands. This review covers various neural network architectures and strateges tailored for TinyML. They discuss efficient neural networks and lightweight architectures like MobileNet and MCUNet designed for resource constrained devices. Then, they discuss model compression techniques such as knowledge distillation, model pruning, and quantization to optimize models while preserving performance. They also discuss deployment challanges such as the the difficulties of deploying deep learning models on ultra-low-power MCUs, including limited memory and processing power. They present the applications of TinyML with MEMS-based applications such as environmental sensing, gesture recognition, and predic-

tive maintenance that benefit from TinyML. They finally discuss the current limitations and future directions of TinyML such as trade-offs between model accuracy and resource usage, the need for specific benchmarks, and potential advancements in hardware and algorithms.

Another extensive review on Tiny Machine Learning is done by Han et al., with a particular focus on its application in low-power, resource-constrained devices like microcontrollers [51]. This systematic review synthesizes 47 publications since 2019 and addresses some key areas of TinyML. It reviews the hardware that is commonly used to implement TinyML. Then it reviews the frameworks that are used to deploy models on embedded devices, such as TensorFlow Lite. Then, some public datasets that are commonly used in TinyML research is given. Then, the use cases and applications of TinyML are discussed, which include keyword spotting, image classification, anomaly detection, and motor control. Then, algorithms such as Convolutional Neural Networks and Deep Neural Networks which are frequently used due to their adaptability are discussed. The review also highlights the challenges in the field, including limited datasets, model interpretability, and the need for hardware-specific optimization. It concludes with recommendations for future research, advocating for expanded datasets and more robust testing to advance TinyML's application in consumer and industrial settings.

There are a numerous other studies on the classification of Parkinson's disease hand tremors using various algorithms from more traditional signal processing algorithms to more novel machine learning algorithms. In this chapter we attempted to explain some of the existing methods that are promising in classifying Parkinson's disease hand tremors. We also present some of the studies in Tables 3.1 and 3.3, which are adapted from Sigcha et al, [46], as a summary of some of the methods in this field.

This literature review highlighted several promising techniques for tremor detection using various methods, including machine learning methods. Moreover, there are various papers that discuss implementing complex machine learning models on embedded devices, with particular focus on optimization methods such as quantization and pruning. However, gaps still remain in real-time time detection of Parkinson's tremor on resource-constrained devices. To address these, we have selected a set of algorithms for evaluation, applied various optimization methods and presented our results, detailed in the following chapters.

Table 3.1: Summary of previous work on Parkinson's tremor detection using wearable sensors (part 1). Adapted from Sigcha et al. (2021) with minor changes.

Study	Year	Sensors and Location	Methods	Main Results and Observations
Patel et al. [52]	2010	8 uniaxial accelerometers (arms and legs)	SVM	Achieved low error in clinical score estimation, indicating potential for ambulatory settings.
Rigas et al. [53]	2012	6 accelerometers (wrists, ankles, sternum, waist)	Hidden Markov Model	High specificity (97%) and sensitivity (95%) in tremor detection, demonstrating capability to differentiate PD symptoms; device count may limit usability.
Roy et al. [54]	2013	EMG and accelerometers (forearms and shanks)	Dynamic neural network	Reported high sensitivity (92.9%) and specificity (90.2%) for tremor detection in unconstrained activities; multi-sensor setup may be challenging for real-world use.
Ahlrichs et al. [55]	2014	Triaxial accelerometer (wrist)	SVM	Frequency-domain features identified as sufficient for tremor detection with high sensitivity (89.4%); setup is somewhat device-specific.

Table 3.2: Summary of previous work on Parkinson's tremor detection using wearable sensors (part 2). Adapted from Sigcha et al. (2021) with minor changes.

Study	Year	Sensors and Location	Methods	Main Results and Observations
Kostikis et al. [56]	2015	Smartphone accelerometer and gyroscope (wrist)	Machine learning	Reported accuracy of 82% for PD patients; simple setup demonstrates potential for consumer-level monitoring solutions.
Braybrook et al. [57]	2016	Triaxial accelerometer (wrist)	Spectral thresholding	Specificity and selectivity around 92%; effective for ambulatory assessment with limited flexibility for non-standard activities.
Jeon et al. [58]	2017	Watch-like device (accelerometer and gyroscope)	Machine learning	Achieved accuracy of 85.5% in tremor scoring; watch-like design could be more user-friendly for patients in daily life.
Kim et al. [59]	2018	Triaxial accelerometer and gyroscope (wrist)	CNN	Achieved accuracy of 85%, indicating deep learning's effectiveness for tremor detection with consumer-grade sensors.

Table 3.3: Summary of previous work on Parkinson's tremor detection using wearable sensors (part 3). Adapted from Sigcha et al. (2021) with minor changes.

Study	Year	Sensors and Location	Methods	Main Results and Observations
Hssayeni et al. [60]	2019	Gyroscope (wrist and ankle)	RNN and Gradient Tree Boosting	High correlation with clinical scores (r=0.93); deep learning approach demonstrated robustness but requires significant computational resources.
Pierleoni et al. [61]	2019	Watch-like device (triaxial accelerometer, gyroscope, magnetometer)	Thresholding	Accuracy of 97.7% for tremor detection; high reliability, though specific to controlled conditions.
Mahadevan et al. [62]	2020	IMU watch (accelerometer)	Threshold and machine learning	High correlation (Pearson = 0.97) with clinical tremor constancy scores; effective but lacks validation in diverse settings.
San Segundo et al. [47]	2020	Triaxial accelerometer and gyroscope (wrist)	CNN	Error lower than 5% is achived when estimating the percantage of the tremor in the laboratory environment.

4 Methods

In this chapter, we present the evaluation metrics that we use in this study. Then, we describe the dataset that we use to train and evaluate our models. Finally, we describe the details of each of the implemented algorithms.

4.1 Evaluation Metrics Used in the Thesis

The performance of classification models is typically evaluated using metrics such as accuracy, precision, recall, and the F1 score. These metrics help determine how well the model can generalize from the training data to unseen test data.

To be able to evaluate the various methods that are implemented for real-time Parkinson's tremor detection on an embedded device, we use a set of performance metrics that reflect both the accuracy and efficiency of the models. Accuracy metrics are used to evaluate the algorithms on how accurately they can predict the tremor, whereas efficiency metrics are used to evaluate the algorithms in terms of model complexity, energy efficiency, and prediction latency. Considering these various metrics together makes it possible to assess trade-offs between them when comparing the algorithms. In this section, the metrics that are used in the evaluation are briefly described and their mathematical descriptions are given.

4.1.1 Common Metrics to Evaluate Machine Learning Algorithms

Accuracy: Accuracy is the ratio of correctly predicted instances to the total instances. It provides a general measure of how well the model is performing.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number of Instances}} \tag{4.1.1}$$

Precision: Precision is the ratio of true positive predictions to the total predicted positives. It indicates how many of the predicted tremor events are actually correct.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.1.2}$$

Recall (Sensitivity): Recall is the ratio of true positive predictions to the total actual positives. It measures the model's ability to detect true tremor events.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.1.3}$$

F1 Score: The F1 Score is the harmonic mean of precision and recall, providing a single metric that balances both concerns.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.1.4}$$

Specificity: Specificity is the ratio of true negative predictions to the total actual negatives. It measures the model's ability to correctly identify non-tremor events.

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}} \tag{4.1.5}$$

4.1.2 Additional Metrics Related to Implementation on Embedded Devices

Total Parameters: Total parameters refer to the number of weights and biases in the model. This metric is critical as it impacts both the computational load and the memory requirements on embedded devices.

$$\text{Total Parameters} = \text{Number of Model Weights and Biases} \tag{4.1.6}$$

Model Size: Model size refers to the total memory usage of the model, including all parameters. This helps determine whether the model can fit within the memory constraints of an embedded device.

$$\text{Model Size} = \text{Total Bytes of Model Parameters} \tag{4.1.7}$$

Total Mult-Adds: This metric measures the total number of multiplications and additions required for a single inference, providing insight into the model's computational efficiency, which is essential for evaluating its suitability for real-time applications.

$$\text{Total Mult-Adds} = \text{Sum of Multiplications and Additions in the Model} \tag{4.1.8}$$

Although this thesis primarily focuses on total parameters, model size, and total mult-adds, these metrics are directly related to throughput, energy consumption, and inference time. By optimizing these metrics, the model can achieve improved throughput (handling more inferences per second), reduced inference time (faster predictions), and lower energy consumption, which are important considerations for real-time applications on embedded devices.

4.2 Description of the Training Dataset

This chapter describes the dataset that is used for training the various ML algorithms that are evaluated in this thesis. The dataset is obtained from the Human Sensing Lab at Carnegie Mellon University. The dataset can be found at `https://humansensinglab.github.io/hs_datasets.github.io/parkinson.html`

The dataset utilized in this study includes three-axis (X, Y, Z) accelerometer data from Axivity AX3 devices worn on both wrists, recorded at a 100 Hz sampling rate, which are then downsampled to 50 Hz. The data collection process received approval from the Carnegie Mellon University Institutional Review Board, adhering to the Helsinki Declaration.

Data was gathered from 12 participants aged between 62 and 85 years, all of whom had been diagnosed with Parkinson's Disease (PD) 2 to 5 years prior. Participants engaged in various daily activities such as making sandwiches, writing, typing, playing chess, and cards, as well as performing motor tasks from Part III of the UPDRS. Sessions were

recorded from multiple angles (frontal, left side, and right side) to avoid occlusion of hand movements. Tremor occurrences in the accelerometer data were identified and labeled by a trained individual, using video recordings to ensure accuracy and consistency.

Further information about the dataset can be found at [47] and [49].

4.3 Baseline Features for Parkinson's Tremor Classification

In this study, we extracted several baseline features from accelerometer data for observing the differences of each of the features for tremor and not tremor cases, and classifying tremor in Parkinson's disease patients. These features are computed in both the time and frequency domains, as detailed below:

4.3.1 Time-Domain Features

The following features were computed from the raw accelerometer signals:

- **Mean**: The mean value of the signal for each axis x, y, and z, which gives a measure of the central tendency of the signal.

$$\text{Mean} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

- **Standard Deviation (STD)**: A measure of the amount of variation or dispersion in the signal around the mean value.

$$\text{STD} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

- **Median**: The median is the middle value of the sorted signal and is less sensitive to outliers than the mean.

$$\text{Median} = \text{Median}(x_1, x_2, \ldots, x_N)$$

- **Max and Min Values**: The maximum and minimum values of the signal, which help capture the amplitude and range of the signal.

$$\text{Max} = \max(x_1, x_2, \ldots, x_N), \quad \text{Min} = \min(x_1, x_2, \ldots, x_N)$$

- **RMS (Root Mean Square)**: The RMS value is a measure of the magnitude of the signal and is calculated for each axis x, y, and z. The RMS provides a useful measure of signal energy.

$$\text{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

  where x_i is the signal amplitude at time i, and N is the number of samples in the window.

- **Skewness**: Skewness measures the asymmetry of the signal distribution. A skewness value near zero indicates a symmetric distribution, while positive or negative values indicate right or left skew, respectively.

$$\text{Skewness} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{x_i - \bar{x}}{\sigma}\right)^3$$

- **Kurtosis**: Kurtosis quantifies the tailedness of the signal distribution, i.e., how much of the signal is concentrated in the tails.

$$\text{Kurtosis} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{x_i - \bar{x}}{\sigma}\right)^4$$

- **Entropy**: Signal entropy measures the complexity and irregularity of the signal, with higher values indicating more complex signals.

$$\text{Entropy} = -\sum p(x_i)\log p(x_i)$$

  where $p(x_i)$ is the probability distribution of the signal values.

4.3.2 Frequency-Domain Features

The following features were computed based on the power spectral density (PSD) of the signal using the Welch method:

- **Power in 0-3 Hz Band**: This feature captures the signal power within the 0-3 Hz frequency range, which typically corresponds to non-tremor activities.

$$P_{0-3} = \sum_{0}^{3} P(f)\, df$$

  where $P(f)$ is the power spectral density at frequency f.

- **Power in 3-9 Hz Band**: This feature captures the signal power within the 3-9 Hz frequency range, typically associated with tremor activity in Parkinson's disease.

$$P_{3-9} = \sum_{3}^{9} P(f)\, df$$

- **Peak Frequency**: The peak frequency is the frequency at which the PSD reaches its maximum value, representing the dominant frequency component in the signal.

$$f_{\text{peak}} = \arg\max P(f)$$

4.4 High-pass Filtering to Remove the Effect of Gravity

Regardless of the classification algorithm used, one of the most common preprocessing steps when working with IMU signals, especially in Parkinson's tremor detection, is the application of a high-pass filter to remove the effect of gravity from the IMU signals and, therefore, to improve the performance of the algorithms. When designing a filter, we should be aware that the filter coefficients are usually quantized when implemented in an embedded device. The quantization of filter coefficients may have the effect of moving the poles and zeros slightly compared to the filter that uses coefficients with floating-point precision. The shifting of the poles might be

acceptable as long as the poles do not move outside the unit circle in the pole-zero plot and the resulting frequency response of the filter is still appropriate for the task.

In this section, we will discuss how the quantization affects the designed filter by observing the frequency responses and pole-zero plots for both the unquantized and quantized filter coefficients for different levels of quantization. Note that since the sampling rate of the IMU sensor is 50 samples/second, the shown impulse responses go from 0 Hz to 25 Hz, consistent with the Nyquist theorem.

The filter was designed as a third-order Butterworth high-pass filter with a cutoff frequency of 1 Hz. The quantization is applied with 8-bit and 16-bit precision, and the effect of quantization on the filters is observed.

Figures 4.1, 4.2, and 4.3 show the frequency and phase responses for the floating-point, 16-bit quantized, and 8-bit quantized filters, respectively.

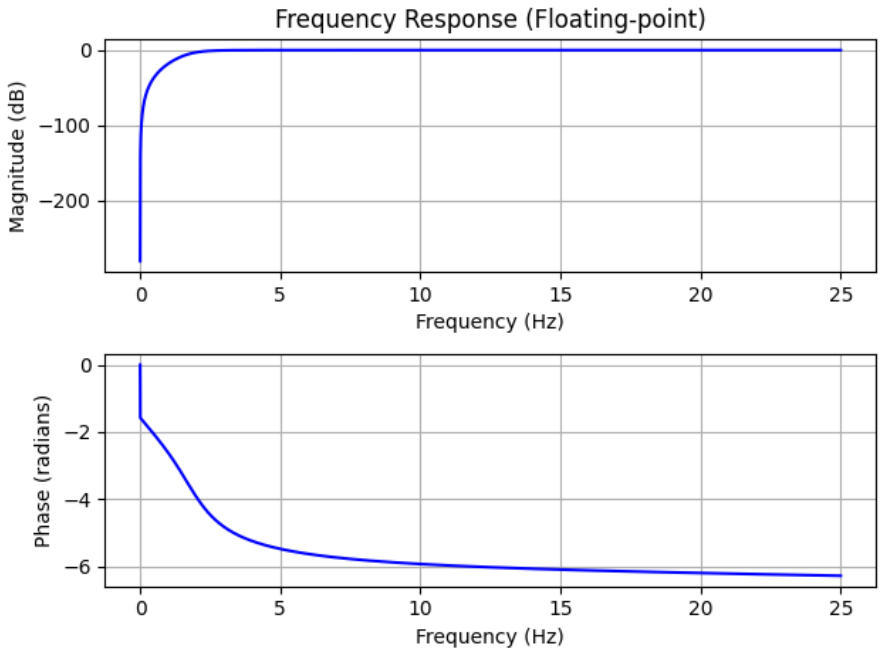

Figure 4.1: Frequency and phase response of the filter (Floating-point).

The pole-zero plots in Figures 4.4, 4.5, and 4.6 illustrate the effects of quantization on the filter's poles and zeros for the floating-point, 16-bit, and 8-bit implementations.

When we observe the frequency responses and the pole-zero plots, even though the frequency responses were altered and the poles and zeros were shifted compared

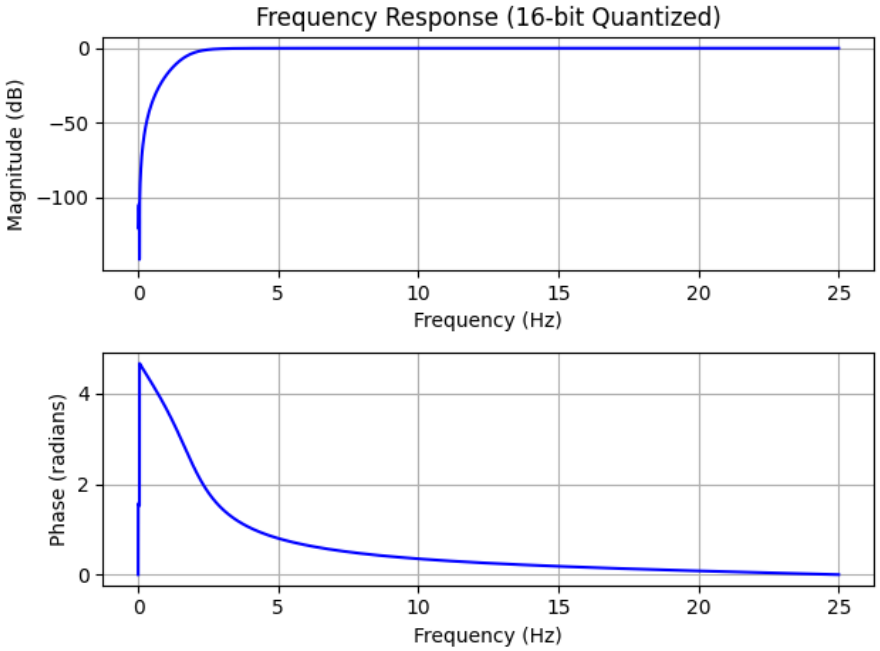

Figure 4.2: Frequency and phase response of the filter (16-bit quantized).

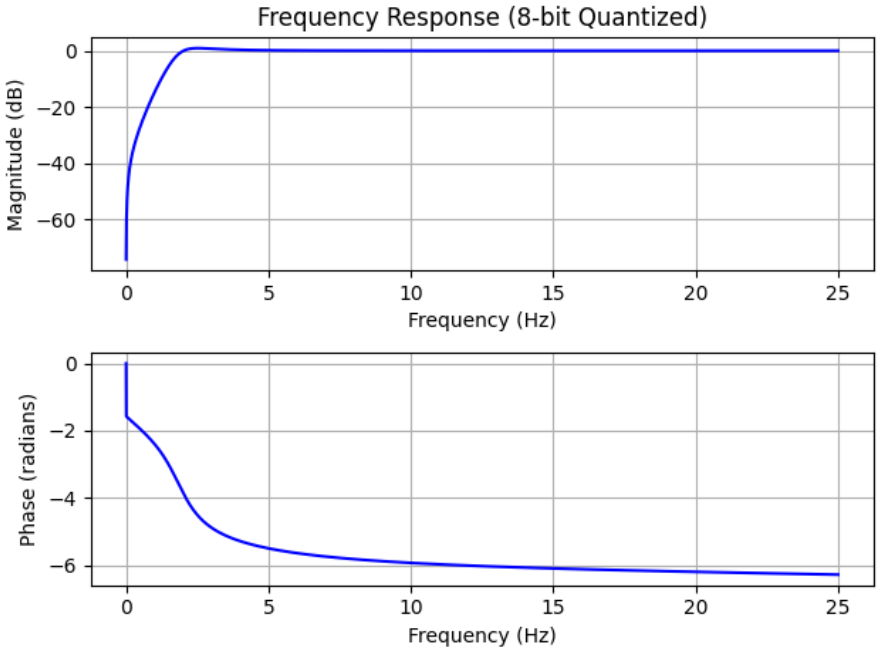

Figure 4.3: Frequency and phase response of the filter (8-bit quantized).

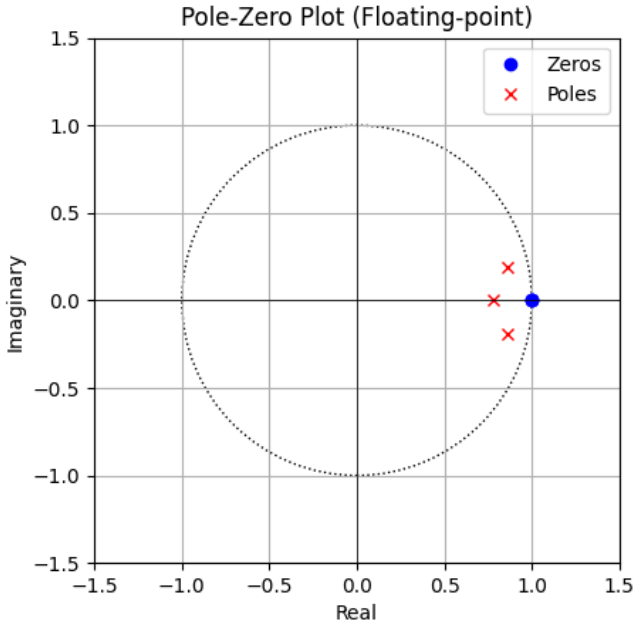

Figure 4.4: Pole-zero plot of the filter (Floating-point).

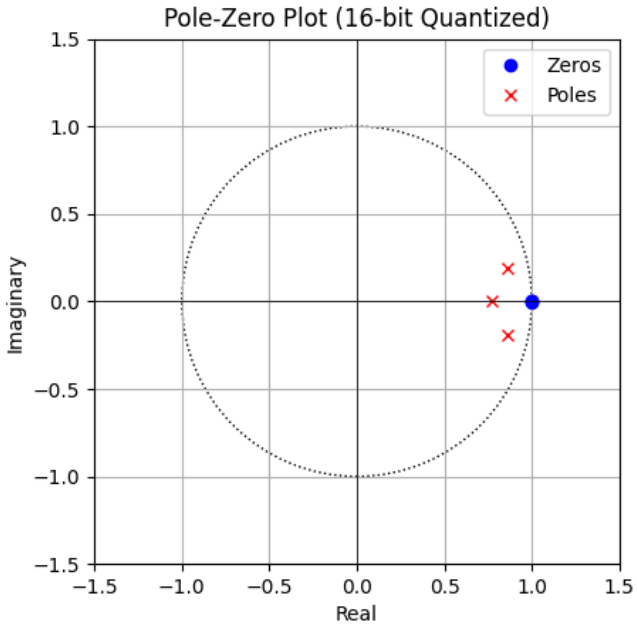

Figure 4.5: Pole-zero plot of the filter (16-bit quantized).

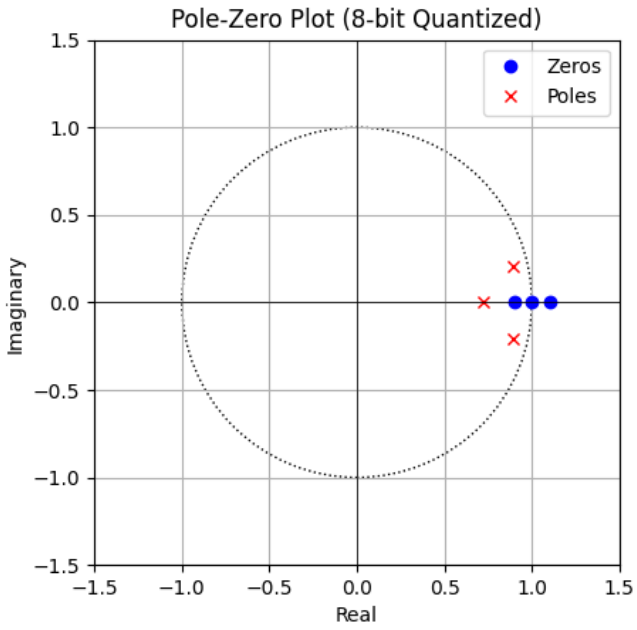

Figure 4.6: Pole-zero plot of the filter (8-bit quantized).

to the filter with an unquantized coefficient, the filters are still able to remove the low-frequency components that result mainly from the effect of gravity on the IMU sensor. We see that the poles are still in the unit circle, meaning that the quantization of the filter coefficients does not lead to unstable filters.

As it can be seen in Figures 4.7 and 4.8 the plots of the signal filtered with 16-bit and 8-bit quantized filter coefficients, filtering with both filters give results very similar to the signal filtered with floating-point unquantized coefficients. All the filters, both quantized and unquantized, where able to remove the effect of gravity, by removing the DC component from the signals, while preserving the shape of the signals. We do not observe any significant difference between the filtered signals, both for the not-tremor and the tremor case.

To conclude, when implementing the filtering on an embedded device as a preprocessing step we can use these filters with 8-bit or 16-bit quantized coefficients, depending on the required accuracy of the filtering step and the available computational power of the embedded device.

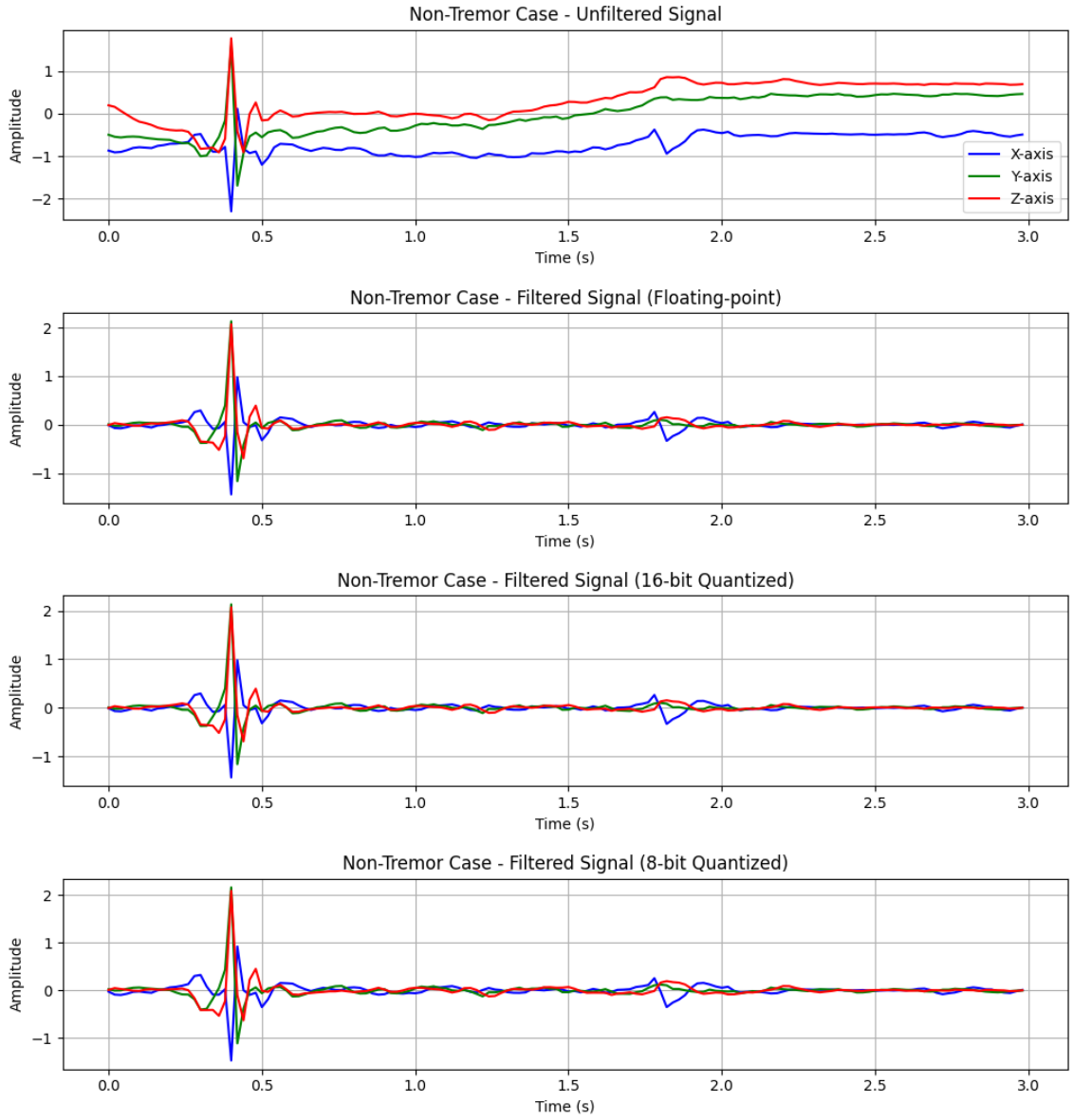

Figure 4.7: Effects of filtering with unquantized and quantized filter coefficients on the not-tremor signal.

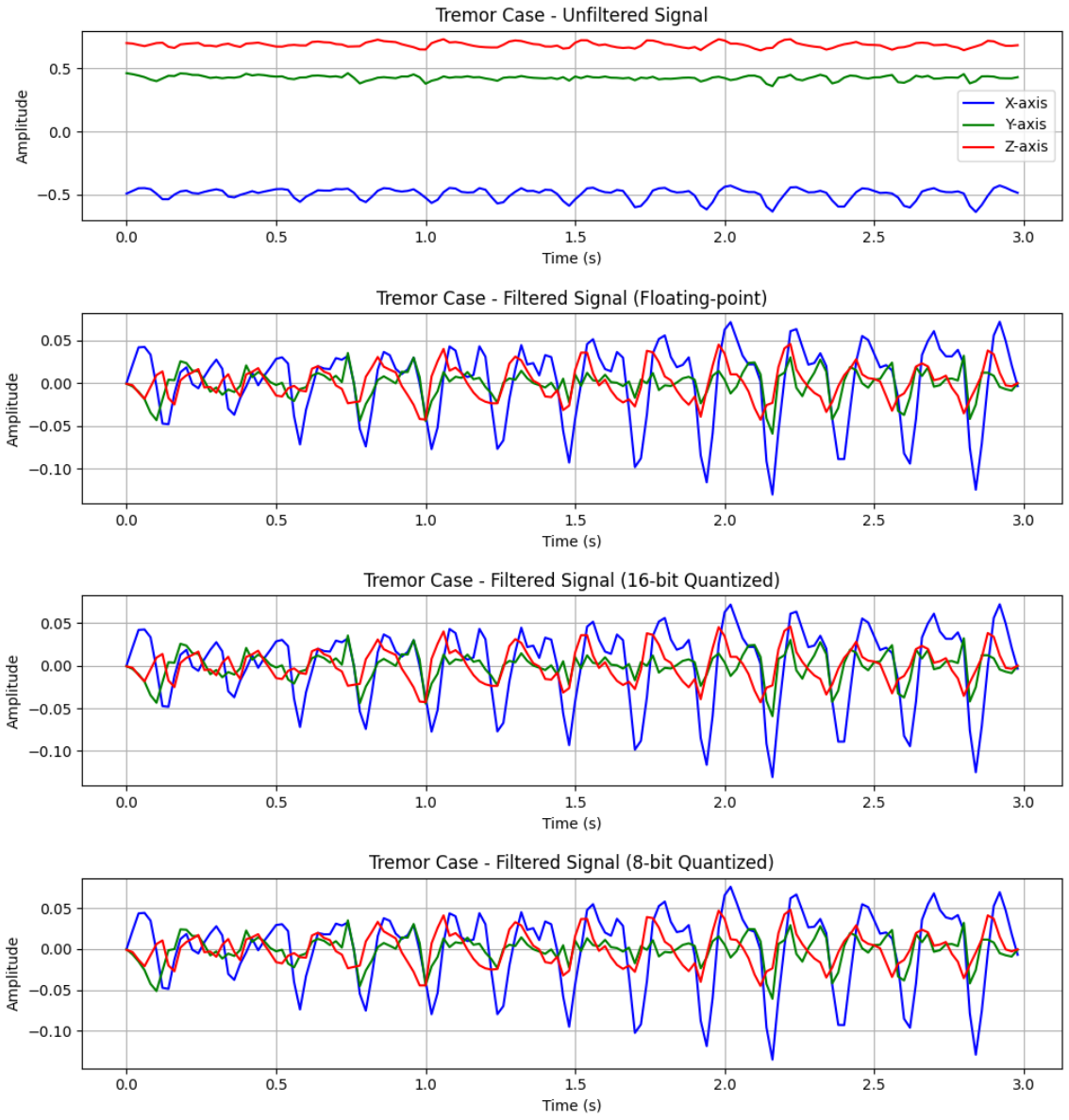

Figure 4.8: Effects of filtering with unquantized and quantized filter coefficients on the tremor signal.

4.5 Implementation of Algorithms

This section describes the implementation of several machine learning models that are used for the initial experiments on detecting Parkinson's tremor using data from wearable Inertial Measurement Unit (IMU) sensors. The models include two Convolutional Neural Networks (CNN), a Support Vector Machine (SVM), a Random Forest (RF), and a Recurrent Neural Network (RNN). Each model is designed to classify tremor and non-tremor movements from IMU data. Data preprocessing, feature extraction, and model architectures are explained in detail in this section. All models are evaluated using Leave One Group Out Cross Validation (LOGO-CV) to ensure robustness and generalizability across different patients.

4.5.1 Data Preprocessing

All models in this study use the same preprocessing steps:

- **High-Pass Filtering**: A Butterworth high-pass filter with a cutoff frequency of 1 Hz and an order of 3 is applied to the IMU data to remove the effect of gravity while maintaining higher frequencies associated with tremor activity.
- **Windowing**: IMU data is segmented into overlapping windows to capture temporal patterns. For the initial experiments, each window contains 150 samples, and adjacent windows overlap by 100 points to ensure sufficient data for capturing movement dynamics. In the later experiments the effect of window lenght is investigated to find a good trade-off point to find a balance between model performance and inference time.
- **Data Augmentation**: To improve model generalization, Gaussian noise addition as an augmentation is applied. The augmentations introduce slight variations to simulate different sensor conditions.

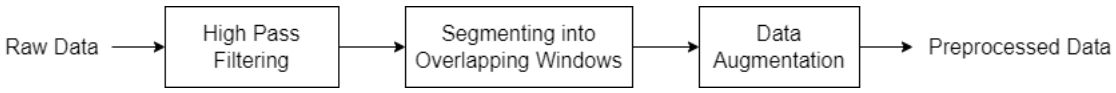

Figure 4.9: Data preprocessing pipeline.

4.5.2 Details of the Implemented Models

Support Vector Machines

The SVM classifier is trained using features extracted from the frequency domain. The key steps in the implementation are as follows:

- **Feature Extraction**: For each window of IMU data, the Fast Fourier Transform (FFT) is applied to compute the energy in specific frequency bands. Energy in the 0Hz-3Hz and 3Hz-9Hz bands is computed for each axis of the IMU data (X, Y, Z).
- **Normalization**: The energy in the 3Hz-9Hz band is normalized by the total energy of the signal to make the features comparable across different windows.
- **Model Training**: An SVM classifier is trained on the extracted features. Leave One Group Out Cross Validation (LOGO-CV) is used, where each group corresponds to a unique patient.

Random Forest

The random forest classifier makes use of the outputs of multiple decision trees to improve classification accuracy. The steps are as follows:

- **Feature Extraction**: Similar to the SVM, the FFT is applied to compute energy in specific frequency bands (0Hz-3Hz and 3Hz-9Hz). These features are then used for classification.
- **Model Training**: A Random Forest classifier with 100 decision trees is trained on the extracted features. Each tree is trained on a subset of the data, and the final classification is based on a majority vote of the trees' predictions.
- **Evaluation**: LOGO-CV is used to evaluate the model's performance, ensuring generalization across different patients.

CNN with 1 Convolutional Layer

CNN with 1 convolutional layer is a Convolutional Neural Network model designed to classify tremors with minimal computational complexity, suitable for deployment on edge devices. The model's architecture is as follows:

- **Input**: The input to the model consists of three channels (X, Y, Z axes of IMU data), with a window size of 150 samples.
- **Convolutional Layer**: A single 1D convolutional layer is applied with 32 filters, a kernel size of 3, and 'same' padding. This layer extracts low-level features from the IMU data, focusing on short-term dependencies between data points.
- **Pooling Layer**: A MaxPooling layer with a pool size of 2 and a stride of 2 is used to reduce the dimensionality of the feature maps, making the network computationally efficient.
- **Fully Connected Layers**: After flattening the pooled feature maps, two fully connected layers are used for classification. The first fully connected layer has 128 neurons, and the second has 64 neurons.
- **Activation and Dropout**: ReLU activation is used for both fully connected layers. To reduce overfitting, dropout with a rate of 0.5 is applied.
- **Output**: A final Softmax layer with 2 output neurons represents the probability of the input belonging to the tremor or non-tremor class.

CNN with 2 Convolutional Layers

CNN with 2 convolutional layers is a more complex architecture compared to CNN with 1 convolutional layer. It gets the same input as the CNN with 1 convolutional layer; however, it uses two convolutional layers to extract deeper and more abstract features from the IMU data:

- **Convolutional Layers**: The first convolutional layer uses 32 filters with a kernel size of 3, followed by a second convolutional layer with 64 filters with a kernel size of 3. Both layers use 'same' padding to preserve the input length.

- **Pooling Layer**: A max pooling layer is applied after each convolutional layer, reducing the dimensionality of the feature maps.
- **Fully Connected Layers**: Similar to CNN with 1 convolutional layer, the network includes two fully connected layers, with 128 and 64 neurons, respectively.
- **Activation and Dropout**: ReLU activation is applied after each fully connected layer, and dropout with a rate of 0.5 is applied to reduce overfitting.
- **Output**: A final softmax layer with 2 neurons provides the probability of tremor versus non-tremor.

Recurrent Neural Network

The Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) units is designed to capture temporal dependencies in the IMU data. The architecture of the model is as follows:

- **LSTM Layers**: Two LSTM layers, each with 64 hidden units, are used to process the IMU data. LSTMs are particularly effective for learning temporal patterns, making them well-suited for time-series data like IMU sensor readings.
- **Fully Connected Layer**: The output from the final LSTM layer is passed through two fully connected layers, with 128 and 64 neurons, similar to the CNN models.
- **Output**: A final softmax layer with 2 neurons provides the probability of tremor versus non-tremor.

4.5.3 Model Training and Evaluation

All models are trained using Leave One Group Out Cross Validation (LOGO-CV) to ensure that the models generalize well to unseen patients. During each training iteration, one group (patient) is held out as the test set, and the remaining groups

are used for training. The following metrics are used to evaluate the models, as previously explained in more detail:

- **Accuracy**: The proportion of windows correctly classified as tremor or non-tremor.
- **Precision**: The proportion of true tremor cases among the predicted tremor cases.
- **Recall**: The proportion of actual tremor cases correctly identified by the model.
- **F1-Score**: The harmonic mean of precision and recall.
- **AUC**: The Area Under the ROC Curve, which summarizes the model's performance across different classification thresholds.

With the methodologies in place, we proceed to evaluate each algorithm's performance across multiple metrics, examining their feasibility for real-time Parkinson's tremor detection on embedded devices.

5 Evaluation of the Implemented Algorithms and Optimization Methods

In this chapter, we present a comparative evaluation of traditional and machine learning algorithms, analyzing their performance in distinguishing Parkinson's tremors. We further investigate the effect of the window size for the IMU data on the performance of the models, to find a balance between accuracy and latency. Moreover, we conduct extensive experiments with different CNN architectures, and apply optimization methods such as pruning and quantization to investigate their effect on the model performance. The results will guide our recommendations for real-time embedded implementations.

5.1 Baseline Feauture Extraction

In this section, we provide the histograms and the correlation matrices for the baseline features that are extracted. To do a comprehensive analysis of the extracted features, the histograms are plotted for each feature and for each axis of the accelerometer data. In this way, we can visually inspect the distribution of the features both for not-tremor and tremor cases. This inspection will give us an idea on how the distribution of various features differ between tremor and not-tremor cases, which will guide our selection of the features that we use for some of the algorithms, such as SVM and random forests. The presented results of the feature extraction can also be used as reference for other studies on Parkinson's tremor detection.

12 features are extracted for each of the 3 axes of the accelerometer data; therefore, we have 36 features in total. The extracted features are mean, standard deviation, median, maximum, minimum, root mean square, skewness, kurtosis, entropy, power in 0-3 Hz and 3-9 Hz frequency bands and the peak frequency. These features are designed to capture important statistical and frequency domain properties of the accelerometer data, which then can be used in distinguishing between not-tremor and tremor movements. The detailed explanation and mathematical description of these features are already given in the methods chapter. The histograms for the

extracted baseline features for X, Y, and Z axes of the accelerometer data are given in Figures 5.1, 5.2, and 5.3, respectively.

We also present the correlation matrices for the extracted baseline features from the X, Y, and Z axes of the accelerometer data in Figures 5.4, 5.5, and 5.6, respectively. A correlation matrix provides a numerical measure of how strongly two features are correlated, ranging from -1 to 1. A correlation value of 1 indicates a perfect positive correlation, meaning that as one feature increases, the other also increases in a proportional manner. Conversely, a correlation value of -1 represents a perfect negative correlation, where one feature increases while the other decreases. A correlation value of 0 implies that there is no relationship between the two features.

When we inspect the histograms, we see that the distributions for all the features have significant overlap between tremor and not-tremor movements, which shows the difficulty of distinguishing between tremor and not-tremor using hand-crafted features. This suggests using neural networks to extract the features from accelerometer signals in a non-linear way might be advantageous and outperform approaches that use hand-crafted features, which is also shown in numerous other studies, as discussed in the literature review chapter.

5.2 Performance Evaluation of the Algorithms without Pruning and Quantization

In this section, we present the initial results of the performance evaluation of various algorithms for Parkinson's tremor detection without applying quantization and pruning. The primary objective is to assess and compare the performance of different models before we move into experimenting with various model optimization methods, such as quantization and pruning. The evaluation includes a simple FFT-based energy thresholding algorithm, traditional machine learning models such as Random Forest Classifier and Support Vector Machine, as well as deep learning models like Convolutional Neural Networks and Recurrent Neural Networks.

We begin by comparing the common performance metrics, such as accuracy, precision, recall, F1 score, specificity, and area under the ROC curve (AUC) for each model. The goal of this comparison is to see how well each model can distinguish

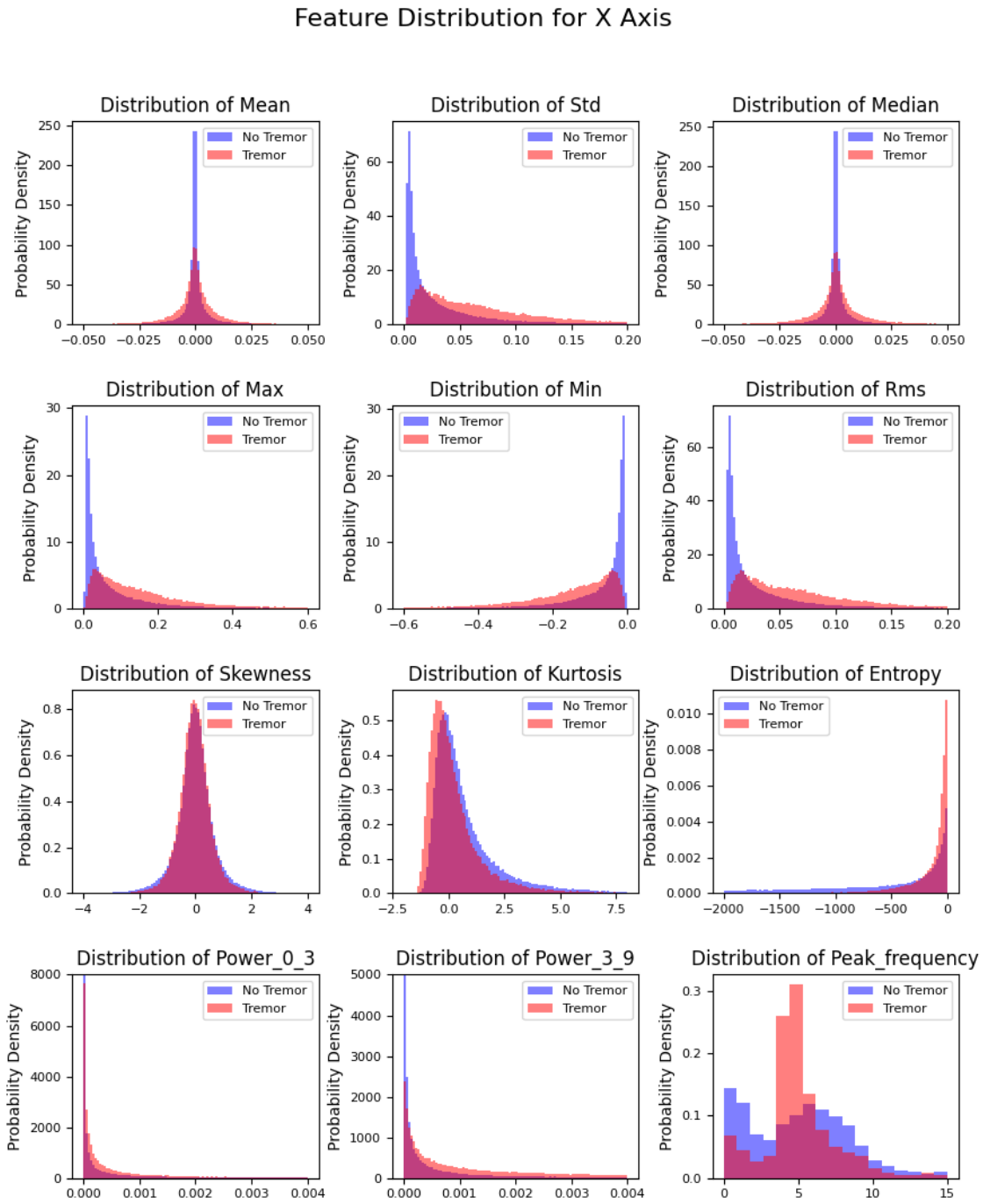

Figure 5.1: Histograms of the baseline features extracted from the x-axis of the accelerometer for tremor and not tremor cases.

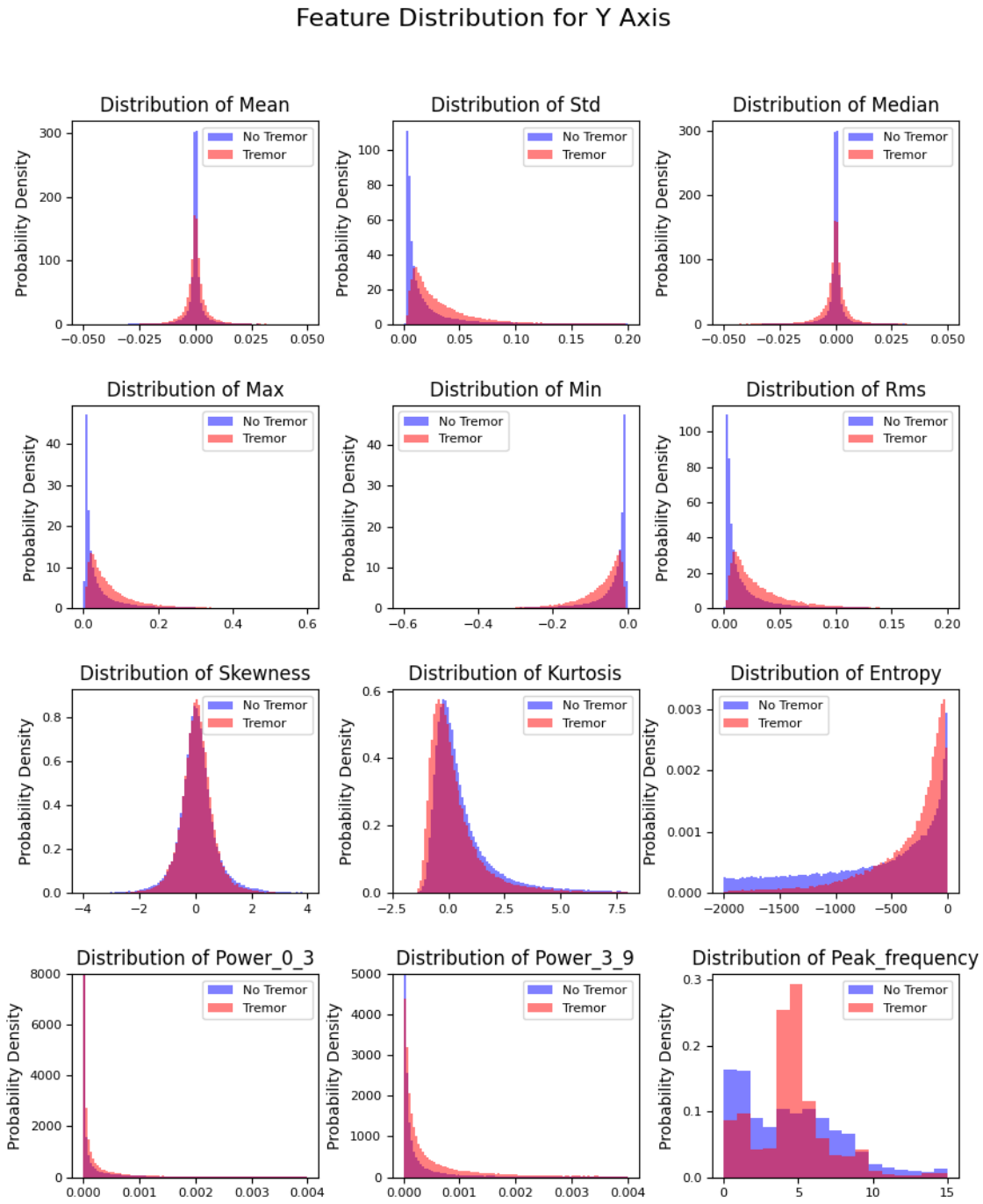

Figure 5.2: Histograms of the baseline features extracted from the y-axis of the accelerometer for tremor and not tremor cases.

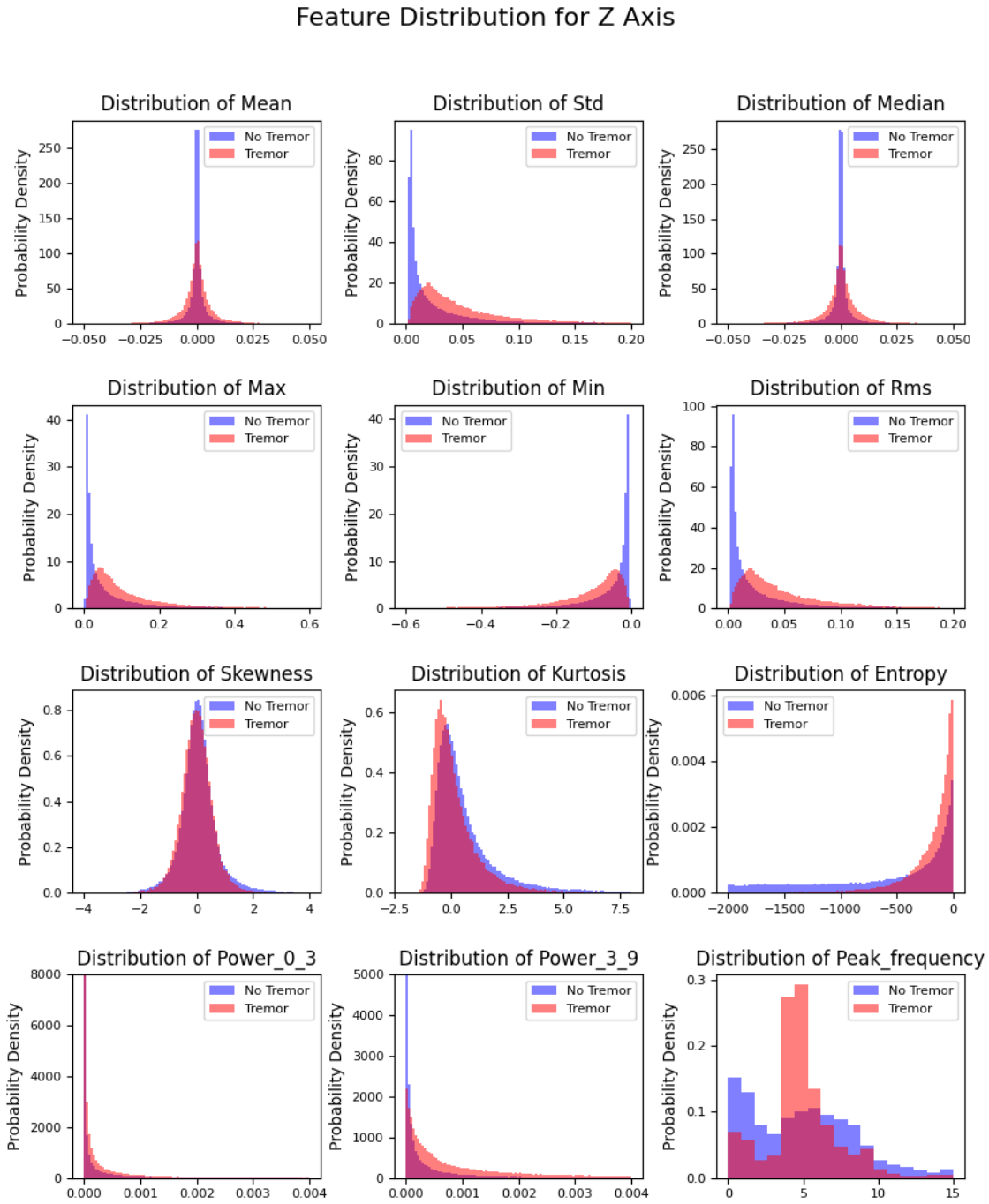

Figure 5.3: Histograms of the baseline features extracted from the z-axis of the accelerometer for tremor and not tremor cases.

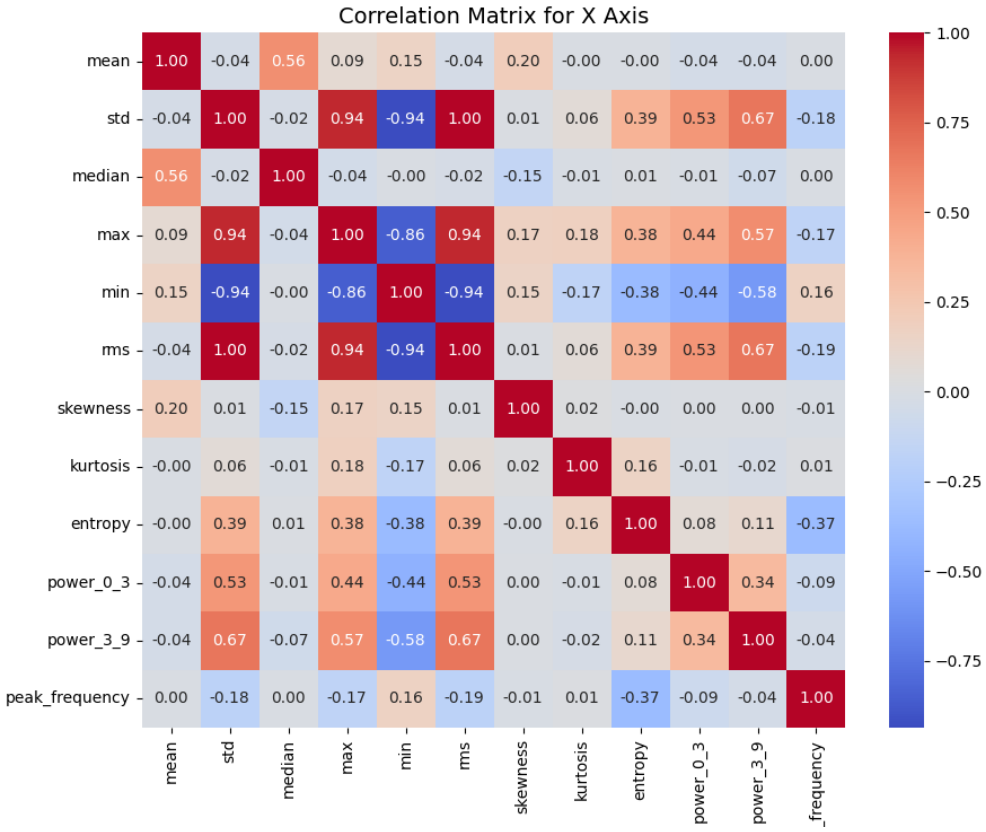

Figure 5.4: Correlation matrix of the baseline features extracted from the x-axis of the accelerometer

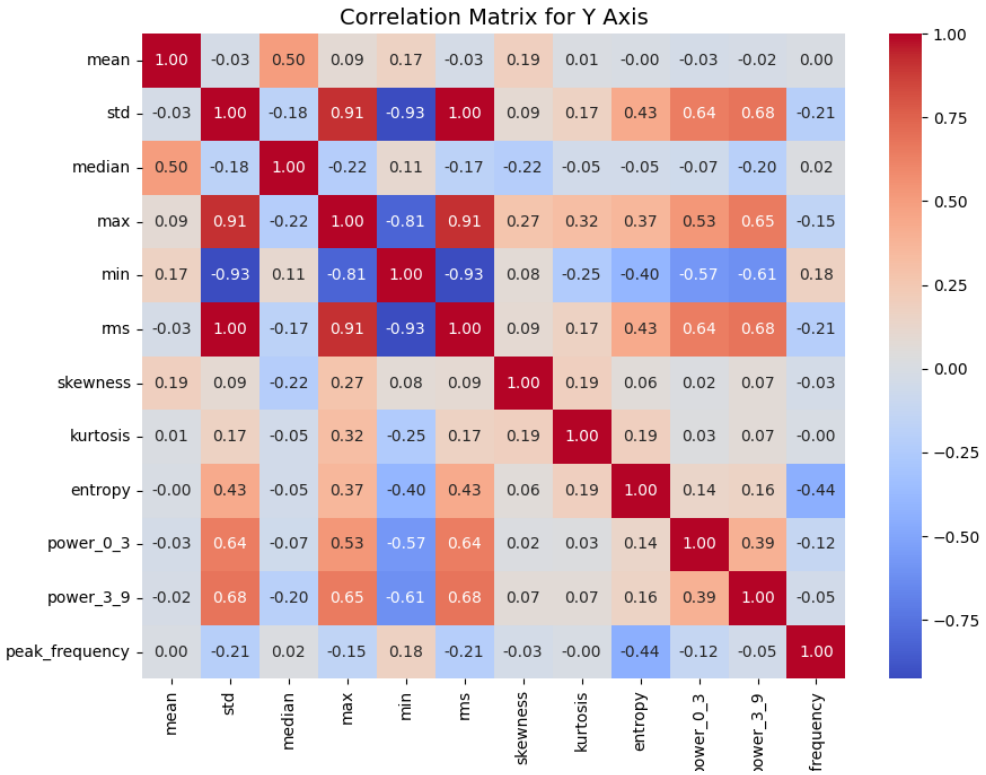

Figure 5.5: Correlation matrix of the baseline features extracted from the y-axis of the accelerometer

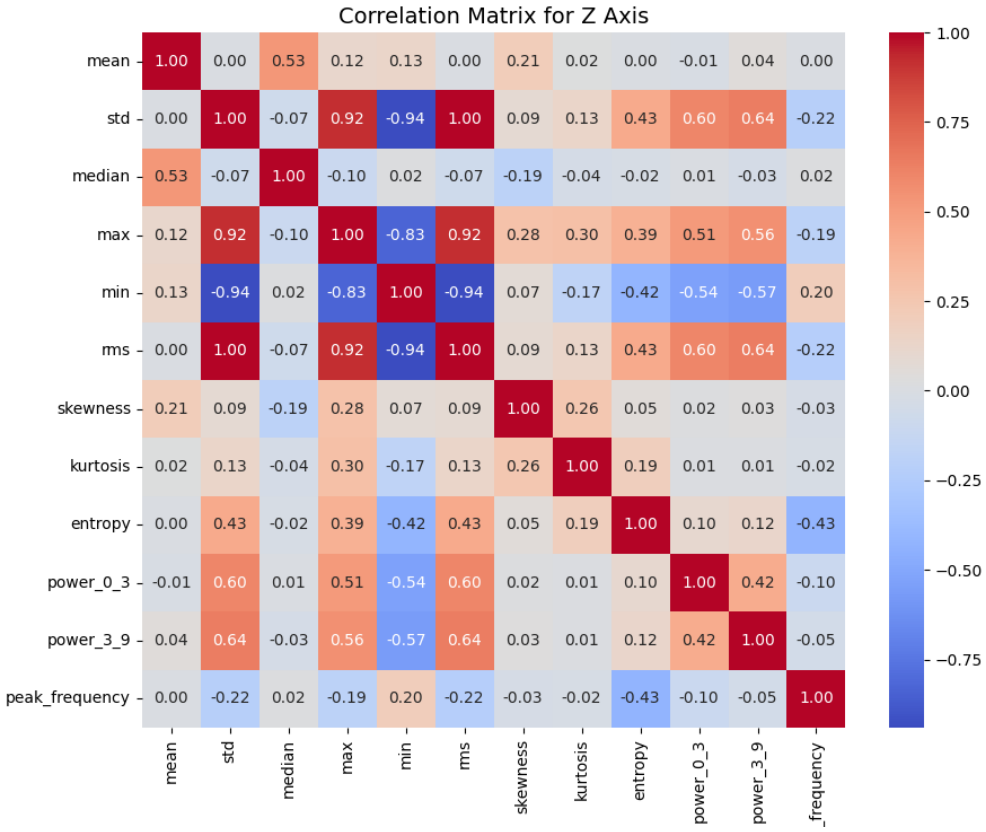

Figure 5.6: Correlation matrix of the baseline features extracted from the z-axis of the accelerometer

between tremor and non-tremor cases, before we move into further experiments with optimization.

After our first evaluation, we experiment with varying window sizes and overlaps of the IMU signal as input to the algoritms on the performance of two selected methods: the FFT-based energy thresholding algorithm and a CNN with one convolutional layer. The results of this evaluation can be used to determine the optimal window lenghth and overlap for the IMU signals as input to the algorithms, to find a good trade-off point that balances between computational efficiency, detection accuracy, and latency. The results of these evaluations are presented in both tables and ROC curves, allowing for a detailed comparison of varying window sizes and overlaps.

5.2.1 Performance Evaluation of Implemented Algorithms

Metric	CNN 1 Conv	CNN 2 Conv	RFC	RNN	SVM	FFT
Accuracy	0.79	**0.82**	0.75	0.76	0.78	0.76
Precision	0.70	0.69	0.60	0.58	**0.74**	0.59
Recall	0.53	**0.69**	0.45	0.66	0.39	0.58
F1 Score	0.60	**0.69**	0.52	0.62	0.51	0.59
Specificity	0.90	0.87	0.87	0.80	**0.94**	0.83
AUC	0.82	**0.86**	0.74	0.80	0.68	0.75

Table 5.1: Performance metrics for various models.

Based on the performance metrics presented in Table 5.1 and the ROC curve shown in Figure 5.7, we can compare the algorithms with each other in terms of various performance metrics and see how well each algorithm is able to distinguish between tremor and not-tremor cases.

As it can be seen from the Table 5.1, the CNN with two convolutional layers achieved the highest accuracy of 82%. This indicates that two convolutional layers makes the model able to extract more complex features in the tremor data, resulting in better overall performance. The CNN with one convolutional layer performed slightly less effectively with an accuracy of 79%. However, it is still a promising model, which shows that, even a simpler CNN architecture with less convolutional layers can effectively extract features and distinguish between tremor and not-tremor cases.

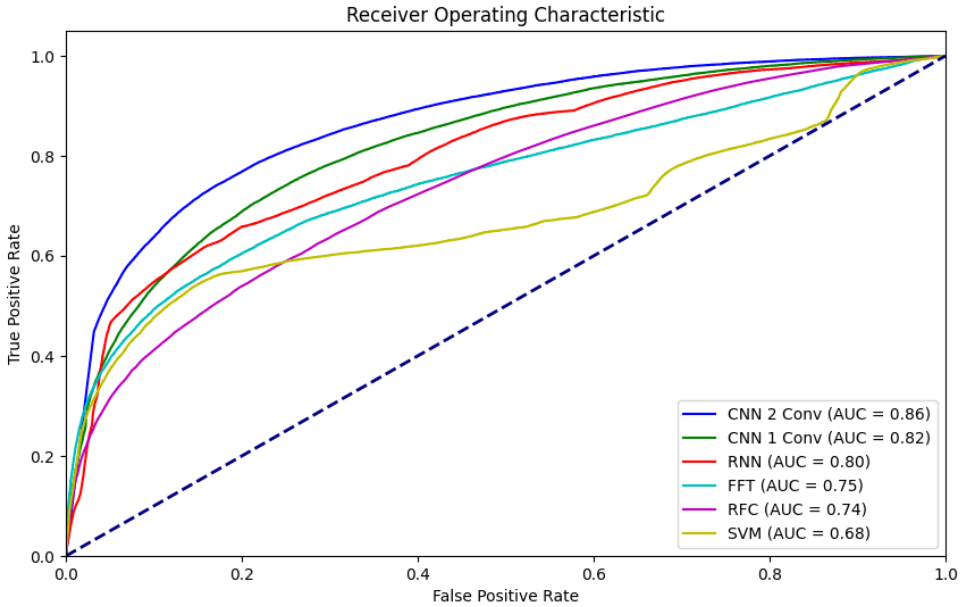

Figure 5.7: ROC Curves Comparison for Different Models

We get the highest precision with the SVM model, with a precision of 74% (Table 5.1). This indicates that the SVM was the most conservative algorithm in making positive predictions, and therefore minimizing false positives. However, we should note that its recall was relatively low at 39%, which means it can not detect a significant amount of true tremor cases.

On the other hand, the CNN models, particularly the one with two convolutional layers, balanced precision and recall more effectively, achieving a precision of 69% and a recall of 69%. Also this balance can be seen at the F1 score, where CNN with two convolutional layers achieved the highest score of 69%.

Specificity is a measure of the ability of the models to correctly identify non-tremor cases. The SVM model had the highest specificity of 94% (Table 5.1), which indicates that the model was successful in avoiding false positives. However, when we consider its overall performance of distinguishing tremor and not-tremor cases by considering its AUC of 68%, the lowest among the algorithms, we can conclude that it was the least successful at differentiating between tremor and not-tremor cases across different decision thresholds.

The CNN with two convolutional layers achieved both the highest accuracy of 82% and the highest AUC of 86%. Figure 5.7, shows that it had the best overall performance in distinguishing between tremor and non-tremor cases. CNN with two convolutional layer has an ROC curve closest to the top-left corner, which reflects a strong trade-off between sensitivity (recall) and specificity. The flatter curve of the SVM model suggests that it was less succesful with this trade-off, which is consistent with its lower AUC as discussed before.

In conclusion, the CNN with two convolutional layers is the most effective model for Parkinson's tremor detection in this study. It offers a good balance between precision, recall, and specificity, and it outperforms other models in terms of overall accuracy and AUC (Table 5.1). However, despite its success in distinguishing between tremor and not-tremor cases, the CNN with two convolutional layers is computationally more demanding to deploy on an embedded device, compared to CNN with 1 convolutional layer. Therefore, we should either reduce its complexity by applying model optimization techniques such as pruning and quantization, or we should choose another simpler model to implement. In the coming section, we continue our experiments on varying window sizes and overlaps with CNN with one convolutional layer, as it has a better balance between performance and complexity. Additionally, the FFT-based energy thresholding method is selected due to its simplicity and efficiency, while still providing reasonable performance in distinguishing tremor and not-tremor cases.

5.2.2 Effect of Varying Window Sizes and Overlaps on FFT Performance

The performance metrics for the FFT-based energy thresholding method across different window sizes and overlaps are presented in Table 5.2, and the corresponding ROC curves are shown in Figure 5.8. These results show how varying window sizes and overlaps affect the accuracy, precision, recall, F1 score, specificity, and AUC of the FFT-based energy thresholding algorithm.

The accuracy of FFT-based energy thresholding algorithm increases with larger window sizes and higher overlaps. The highest accuracy of 78% was achieved with a window size of 150 samples and an overlap of 50 samples (Table 5.2). This shows that larger windows enable the model to make better predictions. The smallest

Metric	30-10	30-20	60-20	60-40	90-30	90-60	120-40	120-80	150-50	150-100
Accuracy	0.68	0.67	0.71	0.70	0.74	0.74	0.76	0.75	**0.78**	0.78
Precision	0.41	0.41	0.44	0.44	0.48	0.48	0.50	0.50	**0.55**	0.54
Recall	0.67	**0.69**	0.64	0.65	0.63	0.62	0.60	0.60	0.55	0.56
F1 Score	0.51	0.51	0.52	0.52	0.54	0.54	**0.55**	0.55	0.55	0.55
Specificity	0.69	0.66	0.73	0.72	0.78	0.78	0.81	0.80	**0.85**	0.85
AUC	0.74	0.74	0.74	0.74	**0.76**	0.76	0.76	0.76	0.75	0.75

Table 5.2: Effect of varying window sizes and overlaps on the performance of the FFT-based Parkinson's tremor detection algorithm. The table presents performance metrics, including Accuracy, Precision, Recall, F1 Score, Specificity, and Area Under the Curve (AUC) for different combinations of window sizes and overlaps. The window sizes (in samples) are shown as the first number in each label, with the second number representing the overlap size.

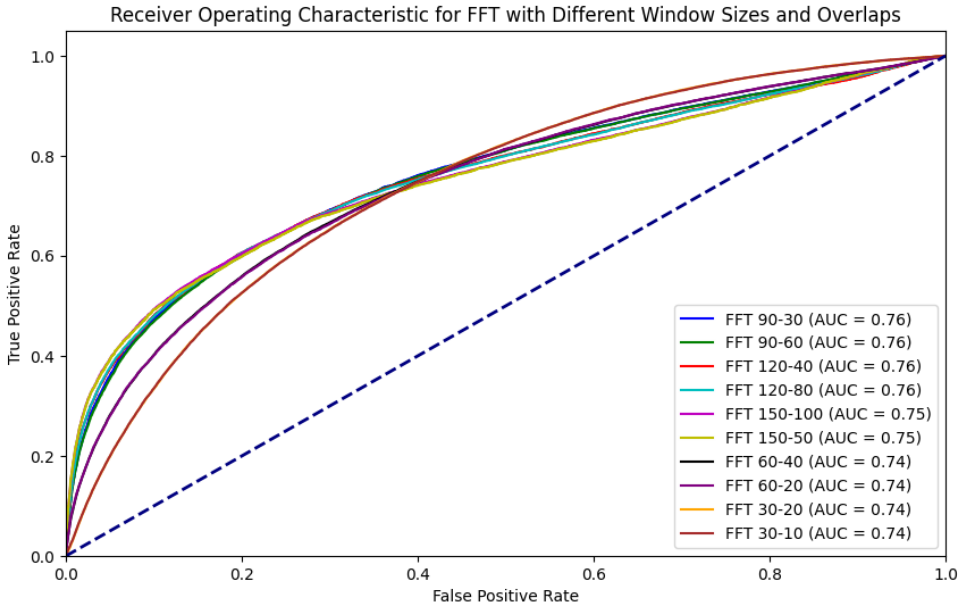

Figure 5.8: ROC Curves for FFT method with varying window sizes and overlaps.

window size (30 samples) with the smallest overlap (10 samples) resulted in the lowest accuracy of 68%, which shows that using smaller window sizes can make the predictions of the algorithms less reliable.

Precision also improved with larger window sizes, reaching its peak of 55% for a window size of 150 samples and an overlap of 50 samples. This trend suggests that the model becomes more conservative, reducing false positives as the window size increases. However, recall does not follow the same trend, peaking at 69% for a window size of 30 samples and an overlap of 20 samples. This discrepancy between precision and recall highlights a trade-off: smaller windows are better at identifying true tremor cases but also produce more false positives.

Specificity and AUC are also positively correlated with larger window sizes. The highest specificity of 85% was observed with a window size of 150 samples and an overlap of 50 samples, indicating that the model was particularly effective in identifying non-tremor cases under this configuration. The AUC values, which measure the overall performance of the model across different decision thresholds, remained relatively stable across the configurations, with the highest AUC of 76% observed for the largest windows (Table 5.2 and Figure 5.8).

The ROC curves in Figure 5.8 reflect the stability of the FFT thresholding method across different configurations. Although there is some variation in the curves, the differences are not substantial, indicating that the FFT method is robust to changes in window size and overlap. The curves for larger window sizes are generally closer to the top-left corner of the plot, confirming that these configurations offer better performance.

In conclusion, the FFT thresholding method demonstrates improved performance with larger window sizes and higher overlaps. This improvement is particularly evident in metrics such as accuracy, precision, and specificity. However, there is a trade-off between precision and recall that must be carefully managed. The consistent AUC values across different configurations suggest that the FFT method is a reliable and robust approach to tremor detection, especially when computational simplicity and efficiency are required.

5.2.3 Effect of Varying Window Sizes and Overlaps on CNN with 1 Convolutional Layer Performance

Metric	30-10	30-20	60-20	60-40	90-30	90-60	120-40	120-80	150-50	150-100
Accuracy	0.74	0.75	0.75	**0.77**	0.76	0.76	0.75	0.77	0.75	0.74
Precision	0.56	0.56	0.57	0.59	0.58	0.59	0.57	**0.60**	0.56	0.56
Recall	**0.73**	0.72	0.68	0.73	0.69	0.72	0.70	0.68	0.64	0.64
F1 Score	0.63	0.63	0.62	**0.65**	0.63	0.65	0.63	0.64	0.60	0.60
Specificity	0.75	0.76	0.78	0.78	0.79	0.78	0.78	**0.81**	0.79	0.79
AUC	0.82	0.82	0.81	**0.84**	0.83	0.83	0.82	0.83	0.80	0.79

Table 5.3: Effect of varying window sizes and overlaps on the performance of the CNN with 1 convolutional layer. The table presents performance metrics, including Accuracy, Precision, Recall, F1 Score, Specificity, and Area Under the Curve (AUC) for different combinations of window sizes and overlaps. The window sizes (in samples) are shown as the first number in each label, with the second number representing the overlap size.

The performance metrics for the CNN with 1 convolutional layer method across different window sizes and overlaps are presented in Table 5.3, while the corresponding ROC curves are shown in Figure 5.9. These results highlight how different configurations impact the accuracy, precision, recall, F1 score, specificity, and AUC of the CNN-based tremor detection model.

The accuracy of the CNN with 1 convolutional layer method shows a marginal improvement as the window size increases, with the highest accuracy of 77% achieved with a window size of 60 samples and an overlap of 40 samples (Table 5.3). This suggests that mid-range window sizes are optimal for capturing sufficient temporal information while avoiding the pitfalls of overly large or small windows. Smaller window sizes, such as 30 samples with a 10-sample overlap, yielded slightly lower accuracy (74%), indicating that they might not capture enough information for effective classification.

Precision generally improves with larger window sizes, peaking at 60% for a window size of 120 samples and an overlap of 80 samples. This indicates that larger windows allow the model to be more conservative, reducing false positives. However, recall is highest at 73% for the smallest window size (30 samples with a 10-sample overlap), highlighting a trade-off: smaller windows are better at identifying true tremor cases but may produce more false positives. As window size increases, recall decreases slightly, reflecting a shift in the model's sensitivity.

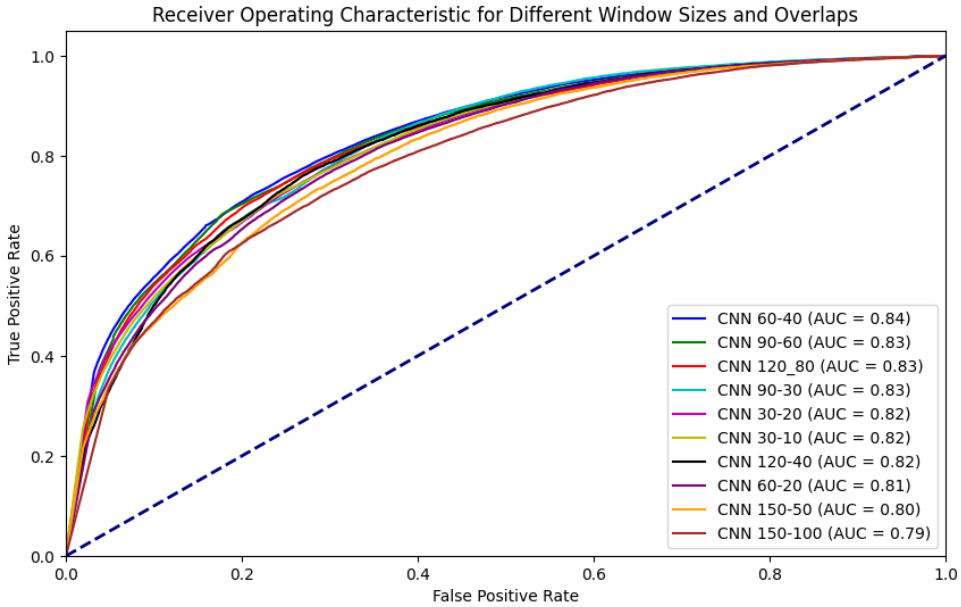

Figure 5.9: ROC Curves for CNN with 1 convolutional layer method with varying window sizes and overlaps.

Specificity and AUC tend to improve with increasing window sizes and overlaps. The highest specificity of 81% was observed with a window size of 120 samples and an overlap of 80 samples, indicating that the model was particularly effective at identifying non-tremor cases under this configuration. The AUC values, which measure the overall performance of the model across different decision thresholds, also reflect this trend, with the highest AUC of 84% observed for a window size of 60 samples and an overlap of 40 samples (Table 5.3 and Figure 5.9).

The ROC curves in Figure 5.9 demonstrate the stability of the CNN with 1 convolutional layer method across different configurations. While there is some variation in the curves, particularly for the smaller and larger windows, the differences are not drastic. The curves for mid-range window sizes (e.g., 60 samples with a 40-sample overlap) are closest to the top-left corner of the plot, indicating the best trade-off between sensitivity (recall) and specificity under these configurations.

In conclusion, the CNN with 1 convolutional layer method shows consistent performance across varying window sizes and overlaps, with mid-range configurations (e.g., 60 samples with a 40-sample overlap) offering the best balance between accu-

racy, precision, and recall. The model's robustness across different configurations, as reflected in the stable AUC values, suggests that it is well-suited for tremor detection tasks. However, there is a notable trade-off between precision and recall that should be considered when selecting the optimal window size for deployment.

5.2.4 Choosing the Window Size and Overlap

Based on the analysis of varying window sizes and overlaps for both the FFT thresholding and CNN with 1 convolutional layer methods, the window size and overlap that provide the best balance between accuracy, computational efficiency, and latency can be identified. For the FFT thresholding method, a window size and overlap of 90-30 or 90-60 offer a good compromise, delivering reasonable accuracy and AUC while reducing latency and computational load. For the CNN with 1 convolutional layer method, the configuration with a window size and overlap of 60-40 emerged as the most balanced, achieving the highest accuracy and AUC while maintaining efficiency and reducing latency. This makes the 60-40 configuration particularly suitable for deployment in resource-constrained environments, such as embedded devices, where real-time processing and efficient use of computational resources are crucial. For the following experiments with different architectures of convolutional neural networks and with optimization methods such as quantization and pruning, we choose to continue with the 60-40 configuration. However, we should note that the selection of the optimum window size and overlap might depend on the specific application and constraints in performance, latency and computational load.

5.3 Choosing the Convolutional Neural Network Architecture

5.3.1 Effect of Model Architecture on Model Size and Computational Load

We have already seen that CNN model was superior to other models in distinguishing tremor from not-tremor. We have also found out that using a window size of 60 samples (around 1 second) has a good trade-off between the classification performance,

latency, and computational efficiency. In this section, we will experiment further with different CNN architectures to understand the trade-offs between the classification performance, model size and total number of multiplication and addition operations needed for inference.

The choice of the CNN architecture is critical, as we aim to optimize various factors at the same time such as the classification performance, the model size, model complexity, and the number of operations needed for inference. In this study, we implemented 12 convolutional neural network architectures, with varying number of convolutional layers and fully connected layers. We vary the number of convolutional layers between 1 to 3 and the number of fully connected layers between 1 to 4. The number of total parameters, the model sizes of the unquantized and quantized models, and the total number of multiplication-addition operations for inference are given in Tables 5.4-5.7. The details of the configurations for each of the architecture are given in the Appendix.

Table 5.4: Total Parameters (in Thousands)

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	41.5	90.8	92.8	93.3
2 Conv	31.3	64.2	66.2	66.7
3 Conv	47.8	72.5	74.5	75.0

Table 5.5: Unquantized Model Size (MB)

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.17	0.36	0.37	0.37
2 Conv	0.14	0.27	0.28	0.28
3 Conv	0.21	0.31	0.32	0.32

Table 5.6: 8-bit Quantized Model Size (MB)

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.04	0.09	0.09	0.09
2 Conv	0.03	0.07	0.07	0.07
3 Conv	0.05	0.08	0.08	0.08

From the Tables 5.4-5.7, we see that it is difficult to draw general conclusions about how the number of convolutional and fully connected layers effect the model size and the number of operations required for inference, since they depend on the

Table 5.7: Total Mult-Adds (in Millions)

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.06	0.11	0.11	0.11
2 Conv	0.16	0.20	0.20	0.20
3 Conv	0.30	0.33	0.33	0.33

specific configuration of the convolutional neural network. Nevertheless, we can do some observations. We observe that adding an extra fully connected layer to the network generally has a more significant effect on the number of total parameters and on the model size, due to its dense configuration. However this effect gets significantly small when the third and fourth fully connected layers are added, since the number of input and output nodes are smaller for these layers compared to the first and second fully connected layers. We also observe that, while the increase in the number of fully connected layers has a smaller effect on the total number of multiplication-addition operations needed for inference, the increase in the number of convolutional layers significantly increase the total number of multiplication and addition operations needed for inference.

To summarize, while it is difficult to draw general conclusions on how to optimize the model size and the total number of multiplication and addition operations needed for inference by only considering the number of fully connected layers and convolutional layers, we should give a special consideration to the number of fully connected layers if we want to minimize the model size and we should give a special consideration to the number of convolutional layers if we want to minimize the total number of operations needed for inference, and therefore optimize the computational load.

In the next section, we train and evaluate the 12 convolutional neural network architectures that are implemented.

5.3.2 Evaluation Results for Various CNN Architectures

In this section, we train and evaluate the CNN architectures that are implemented with varying number of convolutional layers and fully connected layers. The main purpose of this section is to investigate how the CNN architecture effects the performance of the CNN, and how should we decide on the CNN architecture considering the trade-offs between model performance, model size and computational efficiency.

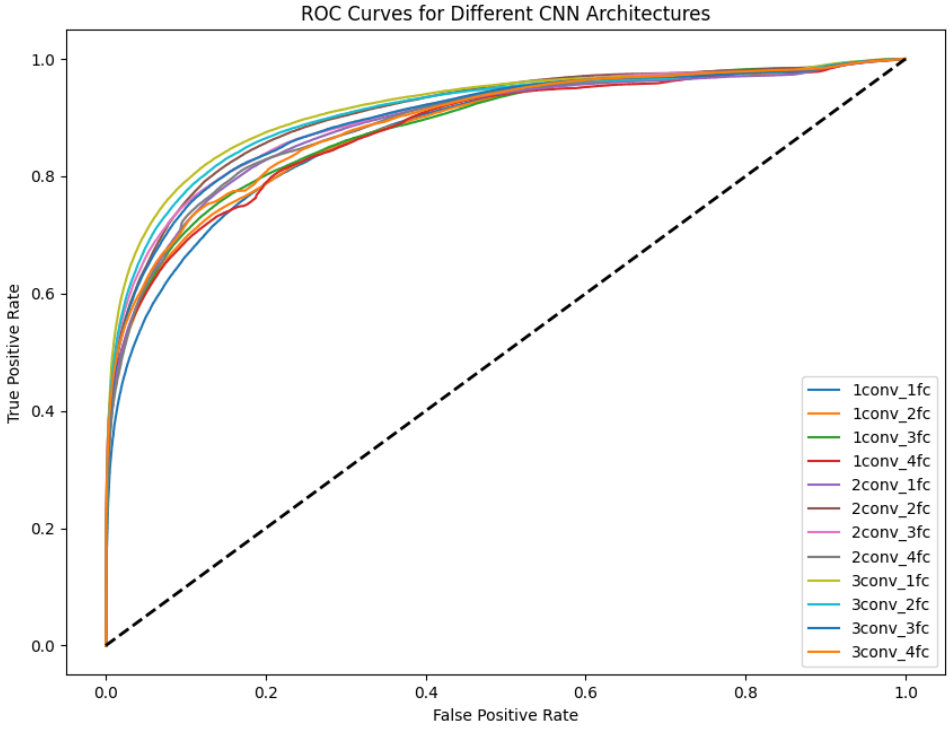

Figure 5.10: ROC Curves for Different CNN Architectures with Varying Number of Convolutional and Fully Connected Layers.

Table 5.8: Accuracy Comparison for Different CNN Architectures

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.8078	0.8375	0.8317	0.8369
2 Conv	0.8426	0.8519	0.8435	0.8416
3 Conv	0.8647	0.8563	0.8463	0.8455

Table 5.9: AUC Comparison for Different CNN Architectures

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.8765	0.8811	0.8836	0.8789
2 Conv	0.8902	0.9081	0.9017	0.8914
3 Conv	0.9166	0.9095	0.8990	0.8906

The performance metrics in Tables 5.8-5.12 indicate that models with three convolutional layers consistently outperform those with fewer layers across all metrics. This improvement suggests that deeper convolutional architectures with more convolutional layers are better equipped to capture the complex patterns in the data, leading to more accurate and balanced tremor classifications. While adding more fully connected layers does provide some performance improvement, especially when going from one to two fully connected layers, the impact diminishes with additional fully connected layers. Particularly, configurations with three convolutional layers and one or two fully connected layers achieve a high AUC and F1 score, which shows strong overall classification ability, while also maintaining high recall and specificity. These results suggest that convolutional depth is crucial for extracting meaningful features, while additional fully connected layers offer limited value beyond a certain point, making the 3 convolution and 1 or 2 fully connected layer models the best performing models in terms of classification performance.

However, as we discussed in the previous section, the more convolutional layers we have, the more computationally demanding the model is. Therefore, to reduce the computational load of the algorithm, without significantly affecting the classification performance, we may prefer one of the 1 or 2 convolutional layer and 1 or 2 fully connected layer configurations. The actual configuration that would be optimum depends on the specific requirements of each study.

The ROC curves in Figure 5.10 again demonstrate that model performance improves with the increasing number of convolutional layers, as configurations with three convolutional layers achieve higher true positive rates for a given false positive

Table 5.10: F1 Score Comparison for Different CNN Architectures

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.6801	0.7030	0.7053	0.6988
2 Conv	0.7202	0.7417	0.7300	0.7215
3 Conv	0.7618	0.7491	0.7317	0.7198

Table 5.11: Recall Comparison for Different CNN Architectures

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.7644	0.7197	0.7537	0.7080
2 Conv	0.7580	0.7955	0.7920	0.7679
3 Conv	0.8096	0.8027	0.7842	0.7423

rate compared to those with one or two layers. This again shows that deeper CNN architectures with more convolutional layers can better capture complex features essential for distinguishing tremor from not-tremor. Adding more fully connected layers beyond two, however, does not improve the performance significantly. Even though configurations with 1 or 2 convolutional layers perform slightly worse than the ones with 3 convolutional layers, we observe that the differences between the ROC curves are not that significant, which indicates that we may safely prefer a smaller CNN model with less number of convolutional layers, therefore optimizing the model size and complexity, without significantly affecting the classification performance.

5.4 Performance Evaluation of the Algorithms with Pruning and Quantization

5.4.1 Evaluation of the Effect of Quantization on CNN

After the initial results we got with the unquantized model, we evaluate the effect of quantization on model performance, by comparing the evaluation results for both unquantized and quantized model.

The worklow for the comparison is given in Figure 5.11. We first preprocess the IMU signals by filtering it with a high pass filter, and windowing it into 1 seconds of windows. For the following sections, we continue with the training of the CNN

Table 5.12: Specificity Comparison for Different CNN Architectures

Conv Layers \FC Layers	1 FC	2 FC	3 FC	4 FC
1 Conv	0.8236	0.8805	0.8601	0.8839
2 Conv	0.8734	0.8724	0.8622	0.8684
3 Conv	0.8848	0.8759	0.8689	0.8832

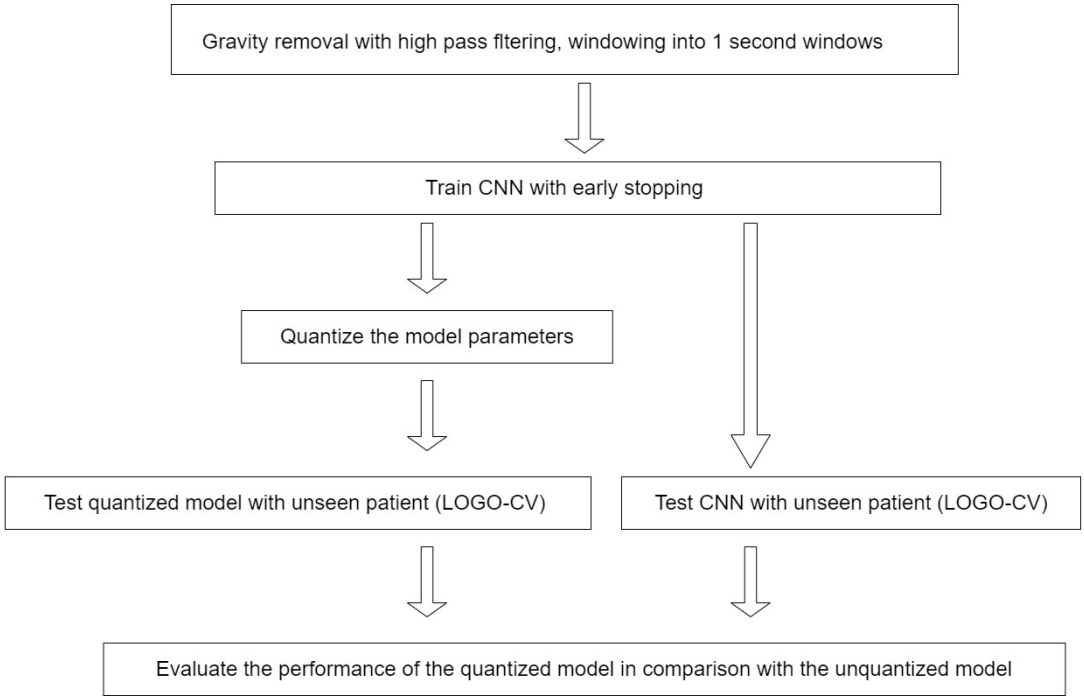

Figure 5.11: Implementation pipeline for the evaluation of the effect of quantization on CNN.

model with 2 convolutional layers and 2 fully connected layers configuration. Then we quantize the model and save the model parameters for both unquantized and quantized model. We use the same test set to get the evaluation results for the two models. Finally, we compare the results with each other to see how quantization affected the performance of the model.

The results show that post-training quantization provides nearly the same performance as the unquantized model. This makes post-training quantization a practical method to optimize the CNN model, especially when the reduction of the model size and the inference speed are critical factors. In this case, post-training quantization almost preserves original model accuracy and class separability, while significantly improving the model size, inference speed and energy efficiency.

Model	AUC	Accuracy	Precision	Recall	F1 Score
Unquantized	0.9077	0.8533	0.7056	0.7742	0.7383
Post-Training Quantized	0.9075	0.8529	0.7047	0.7735	0.7375

Table 5.13: Performance metrics comparison between Unquantized and Dynamically Quantized models

Quantization aware training involves additional training with simulated quantization during forward and backward passes, requiring significant computational resources and implementation effort. This extra effort may be justified if the performance of the post-training quantization is not good enough and the model's architecture is particularly sensitive to quantization. In our case, given the strong performance of post-training quantization, obtaining results for quantization-aware training is not necessary.

5.4.2 Evaluation of the Effect of Unstructured Pruning on CNN

In this study, pruning was employed as a technique to reduce the model's complexity while maintaining its classification performance. The primary goal of pruning is to eliminate redundant or less important weights within the neural network, thereby producing a smaller model without significant decrease in the classification performance. This approach can potentially enhance the efficiency of the model in terms of memory usage and inference speed, which is crucial for deployment on resource-constrained devices.

The pruning strategy used in this research was global unstructured pruning based on L1-norm. In global unstructured pruning, the weights with the smallest L1-norm are pruned across all layers, rather than targeting specific layers individually. This approach allows for a more uniform reduction in model size, where the least important weights (with the smallest magnitude) are removed irrespective of the layer they belong to. Pruning levels were applied incrementally, ranging from 0.0 (no pruning) to 1.0 (complete pruning), to assess the impact on model performance at various sparsity levels.

Following each pruning step, the model was evaluated on the test dataset to measure its classification performance. The metrics considered for evaluation are AUC, accuracy, precision, recall, and F1 score. Additionally, the pruned models were not

fine-tuned or retrained after pruning, in order to observe the direct effects of pruning on model performance.

The results of pruning are summarized in Figure 5.12, which depicts the changes in evaluation metrics as a function of pruning level. The key observations and interpretations are as follows:

At pruning levels up to 0.6, the model's performance remained relatively stable across all metrics. This indicates that the model had a significant amount of redundancy in its weights, allowing for the removal of non-essential parameters without a noticeable loss in predictive accuracy. In this range, pruning primarily targeted weights that contributed little to the network's overall functionality, preserving the model's ability to classify tremor effectively.

A distinct drop in performance metrics occurred around a pruning level of 0.7. This sharp decline suggests that the model had reached a critical threshold, beyond which important weights that contributed to the network's decision-making process were pruned. The network's capacity to generalize effectively was compromised, leading to significant reductions in recall, precision, and F1 score. The sudden performance degradation indicates that the model's structure became too sparse to maintain its original function.

At pruning levels greater than 0.8, there was a rapid decline in all metrics, with some dropping to near-zero values. This behavior indicates that the model was essentially non-functional at these high pruning levels, as the majority of its weights had been removed. The network's ability to learn meaningful patterns was effectively lost due to the extreme sparsity. The sharp drop in metrics demonstrates that aggressive pruning leads to the complete disruption of the model's ability to classify tremor.

An unexpected spike in precision was observed at the 0.8 pruning level. This may be attributed to the model becoming overly conservative in its predictions, resulting in fewer predicted positives but with a higher proportion of true positives. Consequently, although precision appeared to increase, it was accompanied by a severe decline in recall, indicating that the model was making very few positive predictions overall, and the increase in precision does not correspond to an increase in classification performance.

The results indicate that unstructered pruning can be effectively applied to reduce the size of neural networks without a significant loss in performance, up to a certain

threshold. For this study, pruning levels up to 0.6 provided a good balance between model compression and classification performance. Beyond this threshold, the effects of pruning became detrimental, and performance degraded rapidly.

These findings suggest that moderate pruning could be beneficial in applications where memory and computational efficiency are critical, such as deployment on embedded systems and edge devices. However, we must be careful when applying unstructured pruning, since the performance may degrade rapidly after a certain point.

After our experiments on the effect of unstructured pruning on the model performance, we further experiment with structured pruning in the next section.

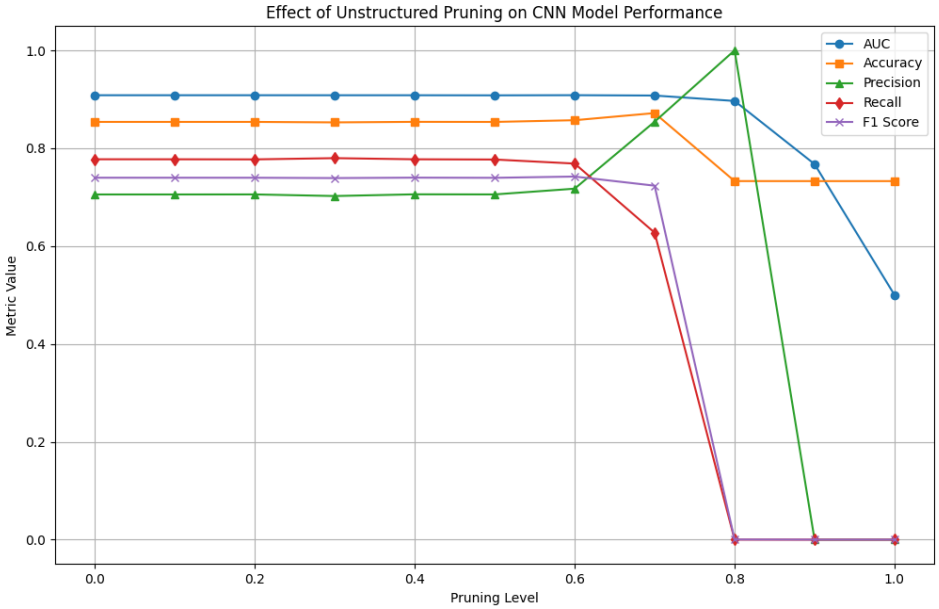

Figure 5.12: Effect of various levels of unstructured pruning on CNN model performance.

5.4.3 Evaluation of the Effect of Structured Pruning on CNN

Structured pruning was applied in this study to reduce the model's complexity by removing entire filters from convolutional layers and neurons from fully connected layers. Unlike unstructured pruning, which eliminates individual weights, structured

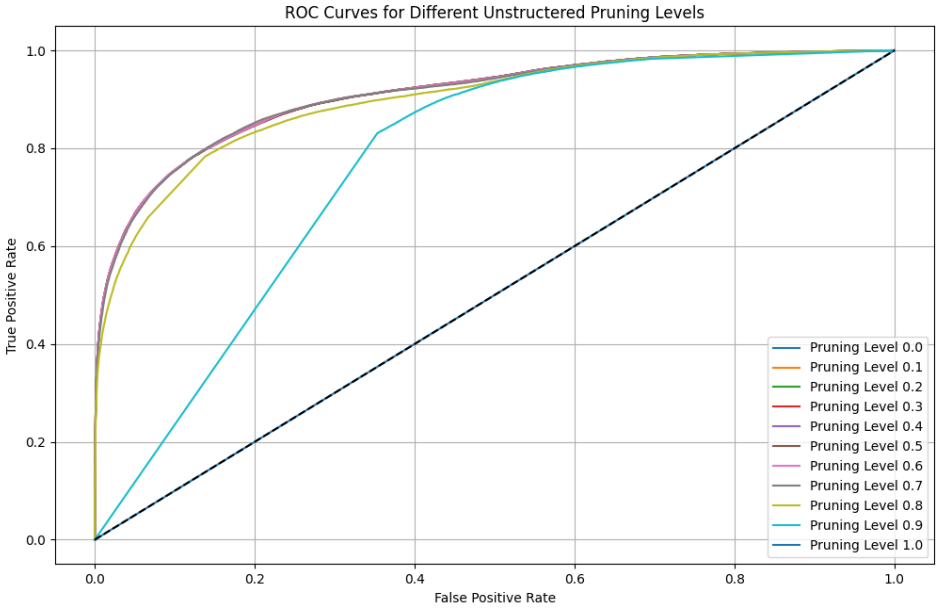

Figure 5.13: ROC Curves for Different Levels of Structured Pruning.

pruning removes entire structures, making the model compression more effective and enabling speedup on hardware platforms. The main advantage of structured pruning is its ability to create a more compact model architecture, leading to potential computational efficiency improvements.

The structured pruning approach used in this study involved `LnStructured` pruning, where entire filters in the convolutional layers and entire neurons in the fully connected layers were pruned based on their L2 norm. The pruning process was applied to the weights of the convolutional layers and the fully connected layers, with pruning levels ranging from 0.0 (no pruning) to 1.0 (complete pruning). The amount of pruning was specified as the fraction of filters or neurons to be removed at each level. Following each pruning step, the model's performance was evaluated on the test dataset using, AUC, accuracy, precision, recall, and F1 score.

The results of structured pruning are depicted in Figure 5.14, which illustrates the changes in performance metrics as a function of pruning level. The key observations and interpretations are as follows:

Pruning Level	AUC	Accuracy	Precision	Recall	F1 Score
0.0	0.9083	0.8538	0.7055	0.7774	0.7397
0.1	0.9083	0.8538	0.7055	0.7774	0.7397
0.2	0.9083	0.8538	0.7056	0.7772	0.7396
0.3	0.9082	0.8528	0.7023	0.7797	0.7390
0.4	0.9082	0.8539	0.7058	0.7774	0.7398
0.5	0.9080	0.8537	0.7055	0.7769	0.7395
0.6	0.9084	0.8572	0.7173	0.7686	0.7421
0.7	0.9076	0.8718	0.8541	0.6274	0.7234
0.8	0.8966	0.7328	1.0000	0.0002	0.0003
0.9	0.7675	0.7328	0.0000	0.0000	0.0000
1.0	0.5000	0.7328	0.0000	0.0000	0.0000

Table 5.14: Performance Metrics at Different Unstructured Pruning Levels

The metrics remained relatively stable up to a pruning level of 0.4, indicating that the model could tolerate structured pruning without a significant impact on its classification performance. This suggests that a substantial portion of the model's filters and neurons were either redundant or contributed minimally to the classification task, allowing for their removal without degrading classification performance of the model.

At pruning levels greater than 0.5, a noticeable drop in most metrics, including precision, recall, and F1 score, was observed. This indicates that the structured pruning process began to remove important filters and neurons, leading to a reduction in the model's ability to distinguish tremor and not-tremor. The sharp decline in performance metrics beyond this threshold suggests that the pruned components were essential for capturing meaningful patterns in the data.

The plot shows a divergence between precision and recall around pruning levels of 0.3 to 0.6. While precision initially increased, recall decreased significantly, indicating that the pruned model was making fewer positive predictions overall, but those predictions were more likely to be correct. This behavior is to the case with unstructured pruning, and the increase in precision does not correspond to an increase in classification performance

Beyond a pruning level of 0.6, all metrics dropped significantly, indicating that the model lost its ability to classify tremor effectively. The complete degradation suggests that the level of structured pruning was too high, removing crucial components

that were necessary for the model to extract and learn the underlying patterns in the data.

The results indicate that structured pruning can be applied effectively to reduce the size of neural networks without substantial performance loss up to a certain threshold. For this study, pruning levels up to 0.4 maintained a balance between model compression and classification performance. Beyond this point, the negative effects of pruning became evident, resulting in a rapid decline in classification metrics.

The findings suggest that moderate levels of structured pruning can be beneficial for achieving model efficiency, especially when computational resources are limited. We must be even more careful when applying structured pruning compared to unstructured pruning, since the degredation of the models classification performance starts at an earlier point.

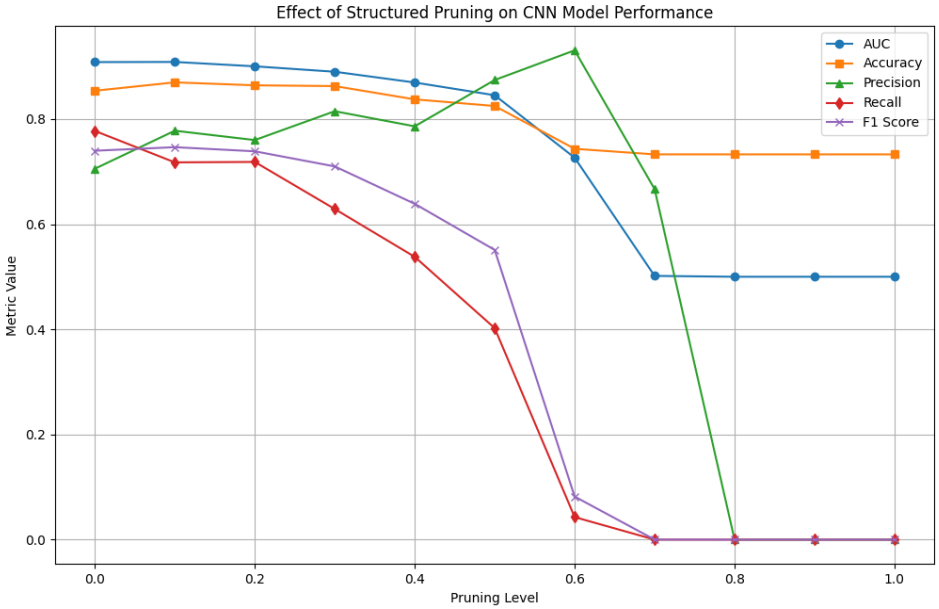

Figure 5.14: Effect of Structured Pruning on Model Performance.

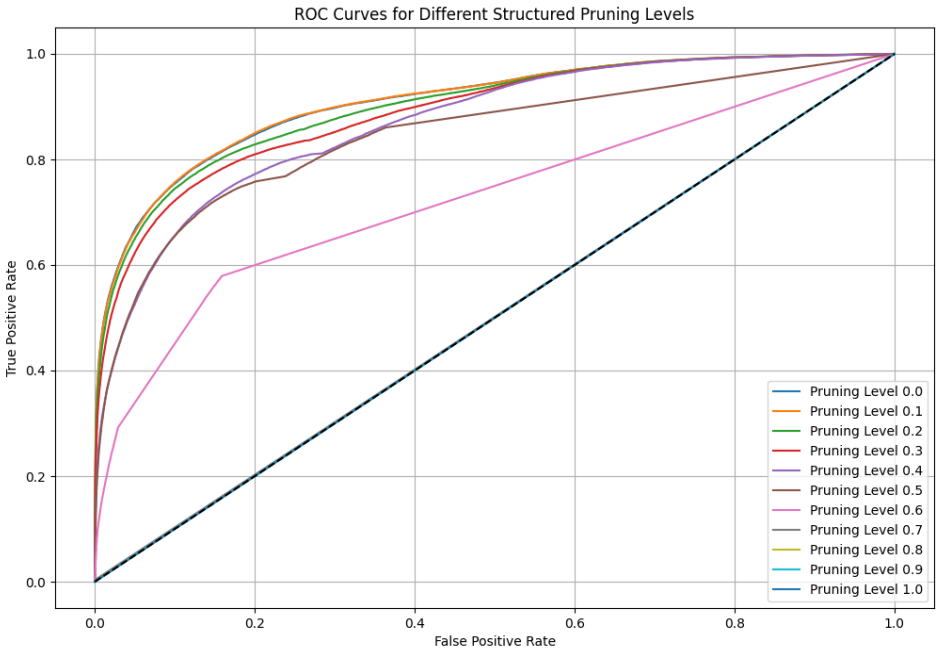

Figure 5.15: ROC Curves for Different Levels of Structured Pruning.

5.4.4 Comparison and Discussion of Unstructured and Structured Pruning

In this section, the main differences and similarities observed between unstructured and structured pruning results are discussed.

In this study, both unstructured and structured pruning approaches were applied to investigate the impact of different pruning techniques and levels on the model's classification performance. Unstructured pruning involves the removal of individual weights with the smallest magnitude (based on the L1-norm) across all layers, while structured pruning removes entire filters from convolutional layers and neurons from fully connected layers. The main goal of these techniques is to reduce the model's complexity and improve efficiency without significantly impacting models classification performance.

The pruning levels ranged from 0.0 (no pruning) to 1.0 (complete pruning). After each pruning step, the models were evaluated using the same set of metrics: AUC, ac-

Pruning Level	AUC	Accuracy	Precision	Recall	F1 Score
0.0	0.9083	0.8538	0.7055	0.7774	0.7397
0.1	0.9085	0.8697	0.7777	0.7176	0.7464
0.2	0.9002	0.8641	0.7600	0.7184	0.7386
0.3	0.8899	0.8626	0.8148	0.6289	0.7099
0.4	0.8694	0.8374	0.7861	0.5379	0.6387
0.5	0.8452	0.8248	0.8742	0.4023	0.5510
0.6	0.7264	0.7434	0.9306	0.0429	0.0819
0.7	0.5016	0.7328	0.6667	0.0000	0.0001
0.8	0.5000	0.7328	0.0000	0.0000	0.0000
0.9	0.5000	0.7328	0.0000	0.0000	0.0000
1.0	0.5000	0.7328	0.0000	0.0000	0.0000

Table 5.15: Performance Metrics at Different Structured Pruning Levels

curacy, precision, recall, and F1 score. The results for both methods are summarized in Figures 5.12 and 5.14.

Both unstructured and structured pruning demonstrated a degree of tolerance to low pruning levels, with metrics remaining relatively stable up to pruning level of 0.6 for the unstructured pruning and up to pruning level of 0.4 for the structured pruning. This suggests that the initial pruning mainly removed redundant or less important weights, filters, or neurons, allowing the models to maintain their classification performance. The performance degradation started at a lower pruning level for structured pruning, indicating that the classification performance of the model is more sensitive to the removal of entire structures.

To conclude, structured pruning was found to be more sensitive to the removal of entire structures from the CNN. While this can lead to higher speedup and compression on hardware platforms, it also makes the classification performance of the model more sensitive to higher levels of pruning. In contrast, the classification performance of the model was less sensitive to unstructured pruning, as it allowed the model to maintain its classification performance even at higher levels of pruning.

The choice of pruning strategy depends on the application requirements. If memory reduction and computational efficiency are critical, structured pruning may be preferred due to its effectiveness in model compression and efficiency. However, if maintaining the classification performance of the model with high pruning levels

is preferred, unstructured pruning could be a better option as it removes individual weights from the CNN, which makes CNN less sensitive to higher levels of pruning.

The results indicate that while structured pruning can achieve better hardware efficiency, it must be applied cautiously. For applications where maintaining classification performance is crucial, structured pruning levels should not exceed 0.4, beyond which significant degradation occurs. Unstructured pruning provides a more flexible approach, as CNN is less sensitive to higher levels of pruning, making it suitable for scenarios where higher levels of pruning is preferred without significant drop in the classification performance.

6 Discussion and Conclusion

This thesis presented a comprehensive evaluation of both traditional signal processing methods and advanced machine learning algorithms, aiming to have a real-time Parkinson's tremor detection algorithm suitable for deployment on embedded devices. The goal of this study was to evaluate and optimize algorithms that balances the trade-offs between the classification performance, latency and computational load at the same time, which are critical factors for wearable devices, where computational and power resources are limited.

The results of the initial evaluation of various algorithms, from more traditional signal processing algorithms to more novel deep learning algorithms, demonstrated that CNNs outperform other methods in terms of classification performance. However, CNNs usually have higher model size and computational requirements than traditional algorithms. Nevertheless, it is possible to deploy CNNs on embedded devices if optimizations are done to decrease to model size and complexity, while maintaining the classification performance. The types of optimizations considered in this study were choosing the optimum CNN architecture, applying neural network quantization and neural network pruning.

In order to choose a CNN architecture that balances the trade-offs between the model size, number of multiplication-addition operations needed for inference and the classification performance, we trained and evaluated 12 CNN architectures with different model sizes and with varying number of convolutional layers and fully connected layers. We observed that increasing the model size does not necessarily improve the classification performance, and smaller models with less convolutional and fully connected layers might be more suitable for embedded device implementations, as they find a balance between the model size, number of multiplication-addition operation needed for inference and the classification performance.

After carefully choosing the CNN architecture that balances between the model size, number of multiplication-addition operations needed for inference and the classification performance, we can further reduce model size and computational load by applying quantization. We demonstrated that it is possible to reduce model size

and computational load of CNNs significantly by applying post-training quantization, while maintaining its classification performance. The effect of the post-training quantization on the classification performance was negligible, which suggest that we should definitely consider applying quantization to CNNs, if we want to deploy them on embedded devices.

After demonstrating the capability of post-training quantization on reducing the model size and computational load while maintaining the classification performance, we conducted extensive experiments with neural network pruning, by applying various levels of pruning to CNN and evaluating it, to observe how does pruning affect the classification performance of the model. We applied two types of pruning to the CNN model: unstructured pruning and structured pruning. Unstructured pruning removed the individual weights from the weight matrix while structured pruning removed entire rows/columns. We observed that the CNN model was less sensitive to unstructured pruning, and it maintained its classification performance up to 60% pruning level, beyond which the classification performance degraded sharply. We further observed that the CNN model was more sensitive to structured pruning, and the decrease in the classification performance started at a much earlier level. However, we discussed that structured pruning is more effective in reducing the computational load, since it removes entire rows/columns from the weight matrix and leads to smaller weight matrices. On the other hand, since unstructured pruning removes individual weights from the weight matrix, it leads to sparse weight matrices and is less effective in achieving high computational gains as compared to structured pruning.

To conclude, this thesis demonstrated that deploying CNN based models for Parkinson's tremor detection on embedded devices is feasible through careful selection of architecture and optimization techniques. While CNNs outperform other algorithms, their typical size and computational demands can limit their direct applicability to resource-constrained environments. However, by implementing optimizations, such as choosing the architecture carefully, and applying quantization and pruning, CNNs can be tailored to operate effectively within the limitations of embedded hardware.

7 Future Work

In this thesis, we demonstrated that CNNs outperform other algorithms which are evaluated in this study. We further showed that CNNs can be deployed in embedded devices if optimized by carefully choosing the architecture and applying neural network quantization and pruning.

There exist opportunities in the future to improve the algorithms that are evaluated in this study. First, the algorithms that are evaluated in this study are not personalized. Future work can focus on model personalization by creating adaptive algorithms that adjust to individual patient profiles. Such personalized models could improve the performance of the algorithms by accounting for each patient's unique tremor characteristics.

Another opportunity to improve the algorithms is by further experimenting with the model optimization techniques, which are crucial when implementing the models on an embedded device. In our work we applied standard quantization and pruning methods, using the relevant functions from the PyTorch library. In the future, more advanced pruning strategies could be evaluated such as dynamic quantization or selective layer pruning, to further improve performance of the models, while keeping the model size small. Furthermore, the quantization and pruning experiments in this study were designed with general edge hardware in mind. In the future, edge hardware specific optimization can be also considered.

There are future possibilities to explore the transfer learning and on-device training. Transfer learning might be useful to train a general model first, on a larger and varied dataset, and then fine tuning it with specific user data to enable the model to learn patient-specific tremor patterns. On-device learning can enable the models to be updated directly on the device to further improve the performance of the algorithms.

In this study, to train and test our algorithms, we only used 3-axis accelerometer data. In the future, different kinds of sensors can be also used and integrated with the acccelerometer data. Integrating data from gyroscopes, magnetometers, and EMG sensors may provide even more information on tremor patterns.

In this study, all the implemented algorithms for real-time Parkinson's tremor detection algorithms were trained and evaluated with saved IMU dataset. Testing the algorithms in the real world with the Parkinson's patients was outside the scope of this study. Future work can also focus on testing these algorithms in clinical/lab environments with Parkinson's patients.

A Appendix

Detailed Convolutional Neural Network Architectures

Table A.1: CNN with 1 Conv Layer and 1 FC Layer

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Linear (2-2)	[-1, 64]	41,024
Dropout (1-3)	[-1, 64]	–
ReLU (1-4)	[-1, 64]	–
Linear (2-3)	[-1, 2]	130
Total params	41,474	
Trainable params	41,474	
Non-trainable params	0	
Total mult-adds (M)	0.06	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.02	
Params size (MB)	0.16	
Estimated Total Size (MB)	0.17	

Table A.2: CNN with 1 Conv Layer and 2 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Linear (2-2)	[-1, 128]	82,048
Dropout (1-3)	[-1, 128]	–
ReLU (1-4)	[-1, 128]	–
Linear (2-3)	[-1, 64]	8,256
Dropout (1-5)	[-1, 64]	–
ReLU (1-6)	[-1, 64]	–
Linear (2-4)	[-1, 2]	130
Total params	90,754	
Trainable params	90,754	
Non-trainable params	0	
Total mult-adds (M)	0.11	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.02	
Params size (MB)	0.35	
Estimated Total Size (MB)	0.36	

Table A.3: CNN with 1 Conv Layer and 3 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Linear (2-2)	[-1, 128]	82,048
Dropout (1-3)	[-1, 128]	–
ReLU (1-4)	[-1, 128]	–
Linear (2-3)	[-1, 64]	8,256
Dropout (1-5)	[-1, 64]	–
ReLU (1-6)	[-1, 64]	–
Linear (2-4)	[-1, 32]	2,080
Dropout (1-7)	[-1, 32]	–
ReLU (1-8)	[-1, 32]	–
Linear (2-5)	[-1, 2]	66
Total params	92,770	
Trainable params	92,770	
Non-trainable params	0	
Total mult-adds (M)	0.11	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.02	
Params size (MB)	0.35	
Estimated Total Size (MB)	0.37	

Table A.4: CNN with 1 Conv Layer and 4 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Linear (2-2)	[-1, 128]	82,048
Dropout (1-3)	[-1, 128]	–
ReLU (1-4)	[-1, 128]	–
Linear (2-3)	[-1, 64]	8,256
Dropout (1-5)	[-1, 64]	–
ReLU (1-6)	[-1, 64]	–
Linear (2-4)	[-1, 32]	2,080
Dropout (1-7)	[-1, 32]	–
ReLU (1-8)	[-1, 32]	–
Linear (2-5)	[-1, 16]	528
Dropout (1-9)	[-1, 16]	–
ReLU (1-10)	[-1, 16]	–
Linear (2-6)	[-1, 2]	34
Total params	93,266	
Trainable params	93,266	
Non-trainable params	0	
Total mult-adds (M)	0.11	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.02	
Params size (MB)	0.36	
Estimated Total Size (MB)	0.37	

Table A.5: CNN with 2 Conv Layers and 1 FC Layer

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Linear (2-3)	[-1, 64]	24,640
Dropout (1-5)	[-1, 64]	–
ReLU (1-6)	[-1, 64]	–
Linear (2-4)	[-1, 2]	130
Total params	31,298	
Trainable params	31,298	
Non-trainable params	0	
Total mult-adds (M)	0.16	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.02	
Params size (MB)	0.12	
Estimated Total Size (MB)	0.14	

Table A.6: CNN with 2 Conv Layers and 2 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Linear (2-3)	[-1, 128]	49,280
Dropout (1-5)	[-1, 128]	–
ReLU (1-6)	[-1, 128]	–
Linear (2-4)	[-1, 64]	8,256
Dropout (1-7)	[-1, 64]	–
ReLU (1-8)	[-1, 64]	–
Linear (2-5)	[-1, 2]	130
Total params	64,194	
Trainable params	64,194	
Non-trainable params	0	
Total mult-adds (M)	0.20	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.03	
Params size (MB)	0.24	
Estimated Total Size (MB)	0.27	

Table A.7: CNN with 2 Conv Layers and 3 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Linear (2-3)	[-1, 128]	49,280
Dropout (1-5)	[-1, 128]	–
ReLU (1-6)	[-1, 128]	–
Linear (2-4)	[-1, 64]	8,256
Dropout (1-7)	[-1, 64]	–
ReLU (1-8)	[-1, 64]	–
Linear (2-5)	[-1, 32]	2,080
Dropout (1-9)	[-1, 32]	–
ReLU (1-10)	[-1, 32]	–
Linear (2-6)	[-1, 2]	66
Total params	66,210	
Trainable params	66,210	
Non-trainable params	0	
Total mult-adds (M)	0.20	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.03	
Params size (MB)	0.25	
Estimated Total Size (MB)	0.28	

Table A.8: CNN with 2 Conv Layers and 4 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Linear (2-3)	[-1, 128]	49,280
Dropout (1-5)	[-1, 128]	–
ReLU (1-6)	[-1, 128]	–
Linear (2-4)	[-1, 64]	8,256
Dropout (1-7)	[-1, 64]	–
ReLU (1-8)	[-1, 64]	–
Linear (2-5)	[-1, 32]	2,080
Dropout (1-9)	[-1, 32]	–
ReLU (1-10)	[-1, 32]	–
Linear (2-6)	[-1, 16]	528
Dropout (1-11)	[-1, 16]	–
ReLU (1-12)	[-1, 16]	–
Linear (2-7)	[-1, 2]	34
Total params	66,706	
Trainable params	66,706	
Non-trainable params	0	
Total mult-adds (M)	0.20	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.03	
Params size (MB)	0.25	
Estimated Total Size (MB)	0.28	

Table A.9: CNN with 3 Conv Layers and 1 FC Layer

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Conv1d (2-3)	[-1, 128, 6]	24,704
ReLU (1-5)	[-1, 128, 6]	–
MaxPool1d (1-6)	[-1, 128, 2]	–
Linear (2-4)	[-1, 64]	16,448
Dropout (1-7)	[-1, 64]	–
ReLU (1-8)	[-1, 64]	–
Linear (2-5)	[-1, 2]	130
Total params	47,810	
Trainable params	47,810	
Non-trainable params	0	
Total mult-adds (M)	0.30	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.03	
Params size (MB)	0.18	
Estimated Total Size (MB)	0.21	

Table A.10: CNN with 3 Conv Layers and 2 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Conv1d (2-3)	[-1, 128, 6]	24,704
ReLU (1-5)	[-1, 128, 6]	–
MaxPool1d (1-6)	[-1, 128, 2]	–
Linear (2-4)	[-1, 128]	32,896
Dropout (1-7)	[-1, 128]	–
ReLU (1-8)	[-1, 128]	–
Linear (2-5)	[-1, 64]	8,256
Dropout (1-9)	[-1, 64]	–
ReLU (1-10)	[-1, 64]	–
Linear (2-6)	[-1, 2]	130
Total params	72,514	
Trainable params	72,514	
Non-trainable params	0	
Total mult-adds (M)	0.33	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.03	
Params size (MB)	0.28	
Estimated Total Size (MB)	0.31	

Table A.11: CNN with 3 Conv Layers and 3 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Conv1d (2-3)	[-1, 128, 6]	24,704
ReLU (1-5)	[-1, 128, 6]	–
MaxPool1d (1-6)	[-1, 128, 2]	–
Linear (2-4)	[-1, 128]	32,896
Dropout (1-7)	[-1, 128]	–
ReLU (1-8)	[-1, 128]	–
Linear (2-5)	[-1, 64]	8,256
Dropout (1-9)	[-1, 64]	–
ReLU (1-10)	[-1, 64]	–
Linear (2-6)	[-1, 32]	2,080
Dropout (1-11)	[-1, 32]	–
ReLU (1-12)	[-1, 32]	–
Linear (2-7)	[-1, 2]	66
Total params	74,530	
Trainable params	74,530	
Non-trainable params	0	
Total mult-adds (M)	0.33	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.03	
Params size (MB)	0.28	
Estimated Total Size (MB)	0.32	

Table A.12: CNN with 3 Conv Layers and 4 FC Layers

Layer (type:depth-idx)	**Output Shape**	**Param #**
Conv1d (2-1)	[-1, 32, 60]	320
ReLU (1-1)	[-1, 32, 60]	–
MaxPool1d (1-2)	[-1, 32, 20]	–
Conv1d (2-2)	[-1, 64, 20]	6,208
ReLU (1-3)	[-1, 64, 20]	–
MaxPool1d (1-4)	[-1, 64, 6]	–
Conv1d (2-3)	[-1, 128, 6]	24,704
ReLU (1-5)	[-1, 128, 6]	–
MaxPool1d (1-6)	[-1, 128, 2]	–
Linear (2-4)	[-1, 128]	32,896
Dropout (1-7)	[-1, 128]	–
ReLU (1-8)	[-1, 128]	–
Linear (2-5)	[-1, 64]	8,256
Dropout (1-9)	[-1, 64]	–
ReLU (1-10)	[-1, 64]	–
Linear (2-6)	[-1, 32]	2,080
Dropout (1-11)	[-1, 32]	–
ReLU (1-12)	[-1, 32]	–
Linear (2-7)	[-1, 16]	528
Dropout (1-13)	[-1, 16]	–
ReLU (1-14)	[-1, 16]	–
Linear (2-8)	[-1, 2]	34
Total params	75,026	
Trainable params	75,026	
Non-trainable params	0	
Total mult-adds (M)	0.33	
Input size (MB)	0.00	
Forward/backward pass size (MB)	0.03	
Params size (MB)	0.29	
Estimated Total Size (MB)	0.32	

List of Abbreviations

AUC	Area Under the Curve
CNN	Convolutional Neural Network
CTFT	Continuous Time Fourier Transform
DFT	Discrete Fourier Transform
Edge AI	Edge Artificial Intelligence
FCN	Fully Connected Network
FFT	Fast Fourier Transform
FPR	False Positive Rate
KNN	K-Nearest Neighbors
LOGO-CV	Leave One Group Out Cross Validation
ML	Machine Learning
PD	Parkinson's Disease
PSD	Power Spectral Density
QAT	Quantization-Aware Training
ROC	Receiver Operating Characteristic
RNN	Recurrent Neural Network
SVM	Support Vector Machine
TinyML	Tiny Machine Learning
TPR	True Positive Rate

A Bibliography

[1] W. Poewe, K. Seppi, C. M. Tanner, *et al.*, "Parkinson disease", *Nature Reviews Disease Primers*, vol. 3, p. 17 013, 2017, Published 23 March 2017. DOI: 10.1038/nrdp.2017.13. [Online]. Available: https://doi.org/10.1038/nrdp.2017.13.

[2] World Health Organization, *Parkinson Disease*, https://www.who.int/news-room/fact-sheets/detail/parkinson-disease, Accessed: 2024-11-07, 2024.

[3] A. Kouli, K. M. Torsney, and W. L. Kuan, "Parkinson's disease: Etiology, neuropathology, and pathogenesis", in *Parkinson's Disease: Pathogenesis and Clinical Aspects*, T. B. Stoker and J. C. Greenland, Eds., Available from: https://www.ncbi.nlm.nih.gov/books/NBK536722/ doi: 10.15586/codonpublications.parkinsonsdisease.2018.ch1, Brisbane (AU): Codon Publications, 2018, ch. 1.

[4] N. Zhao, Y. Yang, L. Zhang, *et al.*, "Quality of life in parkinson's disease: A systematic review and meta-analysis of comparative studies", *CNS Neuroscience Therapeutics*, vol. 27, no. 3, pp. 270–279, 2021, Epub 2020 Dec 28. DOI: 10.1111/cns.13549. [Online]. Available: https://doi.org/10.1111/cns.13549.

[5] C. F. Pasluosta, H. Gassner, J. Winkler, J. Klucken, and B. M. Eskofier, "An emerging era in the management of parkinson's disease: Wearable technologies and the internet of things", *IEEE Journal of Biomedical and Health Informatics*, vol. 19, no. 6, pp. 1873–1881, 2015. DOI: 10.1109/JBHI.2015.2461555.

[6] J. Mei, C. Desrosiers, and J. Frasnelli, "Machine learning for the diagnosis of parkinson's disease: A review of literature", *Frontiers in Aging Neuroscience*, vol. 13, p. 633 752, 2021. DOI: 10.3389/fnagi.2021.633752.

[7] L. V. Kalia and A. E. Lang, "Parkinson's disease", *Lancet*, vol. 386, no. 9996, pp. 896–912, 2015, Epub 2015 Apr 19. DOI: 10.1016/S0140-6736(14)61393-3. [Online]. Available: https://doi.org/10.1016/S0140-6736(14)61393-3.

[8] A. W. Willis, E. Roberts, J. C. Beck, *et al.*, "Incidence of parkinson disease in north america", *npj Parkinson's Disease*, vol. 8, p. 170, 2022, Received 22 March 2022; Accepted 11 October 2022; Published 15 December 2022. DOI: 10.1038/s41531-022-00410-y. [Online]. Available: https://doi.org/10.1038/s41531-022-00410-y.

[9] R. L. Nussbaum and C. E. Ellis, "Alzheimer's disease and parkinson's disease", *New England Journal of Medicine*, vol. 348, no. 14, pp. 1356–1364, 2003, [PubMed] [Reference list]. DOI: 10.1056/NEJM2003ra020003. [Online]. Available: https://doi.org/10.1056/NEJM2003ra020003.

[10] A. H. Abusrair, W. Elsekaily, and S. Bohlega, "Tremor in parkinson's disease: From pathophysiology to advanced therapies", *Tremor and Other Hyperkinetic Movements (New York)*, vol. 12, p. 29, 2022. DOI: 10.5334/tohm.712.

[11] H. Zach, M. Dirkx, B. Bloem, *et al.*, "The clinical evaluation of parkinson's tremor", *Journal of Parkinson's Disease*, vol. 5, pp. 471–474, 2015. DOI: 10.3233/JPD-150650.

[12] S. Sharma and S. Pandey, "Approach to a tremor patient", *Annals of Indian Academy of Neurology*, vol. 19, no. 4, pp. 433–443, 2016. DOI: 10.4103/0972-2327.194409.

[13] Labiotech.eu. "Axovant turns to gene therapy to tackle parkinson's disease". Accessed: September 12, 2024. (2018), [Online]. Available: https://www.labiotech.eu/trends-news/axovant-parkinsons-disease-gene/.

[14] N. Surgery, *Common symptoms of parkinson's disease*, Accessed: 2024-01-31, 2013. [Online]. Available: https://neurologicalsurgery.in/disease/parkinsons-disease/symptoms/.

[15] R. C. Helmich, M. Hallett, G. Deuschl, I. Toni, and B. R. Bloem, "Cerebral causes and consequences of parkinsonian resting tremor: A tale of two circuits?", *Brain: A Journal of Neurology*, vol. 135, no. 11, pp. 3206–3226, 2012. DOI: 10.1093/brain/aws023. [Online]. Available: https://doi.org/10.1093/brain/aws023.

[16] R. N. Bracewell, *The Fourier transform and its applications*. McGraw-Hill, 1986.

[17] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-time signal processing*. Prentice Hall, 1999.

[18] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series", *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[19] P. Heckbert, *Fourier transforms and the fast fourier transform (fft) algorithm*, Revised Jan. 27, 1998, 1998. [Online]. Available: https://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/www/notes/fourier/fourier.pdf.

[20] P. D. Welch, "The use of fast fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms", *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 70–73, 1967.

[21] K. P. Kording, A. S. Benjamin, R. Farhoodi, *et al.*, "The roles of machine learning in biomedical science", in *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2017 Symposium*, Washington, DC: National Academies Press (US), 2018. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK481619/.

[22] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York: Springer, 2006.

[23] A. Maier, C. Syben, T. Lasser, and C. Riess, "A gentle introduction to deep learning in medical image processing", *Zeitschrift für Medizinische Physik*, vol. 29, no. 2, pp. 86–101, 2019, ISSN: 0939-3889. DOI: 10.1016/j.zemedi.2018.12.003. [Online]. Available: https://doi.org/10.1016/j.zemedi.2018.12.003.

[24] H. S. N. S. R. D. Niemann Hornegger and Maier, *Pattern recognition lecture slides*, Licensed under CC BY 4.0, 2022. [Online]. Available: https://lme.tf.fau.de/teaching/.

[25] C. Cortes and V. Vapnik, "Support-vector networks", *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[26] N. Cristianini and J. Shawe-Taylor, *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[27] L. Breiman, "Random forests", *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[28] W. contributors, *Random forest — wikipedia, die freie enzyklopädie*, [Online; accessed 7-November-2024], 2024. [Online]. Available: https://de.wikipedia.org/wiki/Random_Forest.

[29] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[30] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[31] A. Maier, Z. Yang, L. Rist, *et al.*, *Lecture notes in deep learnings*, https://lme.tf.fau.de/category/lecture-notes/lecture-notes-dl/, Lecture Notes, accessed: 2024-10-14, 2021.

[32] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines", in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[33] S. Saha, *A comprehensive guide to convolutional neural networks — the eli5 way*, https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, Accessed: 2024-10-14, 2018.

[34] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors", in *arXiv preprint arXiv:1207.0580*, 2012.

[35] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization", *arXiv preprint arXiv:2106.08295*, 2021.

[36] H. Cheng, M. Zhang, and J. Q. Shi, "A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations", *arXiv preprint arXiv:2308.06767*, 2024.

[37] M. T. Lê, P. Wolinski, and J. Arbel, "Efficient neural networks for tiny machine learning: A comprehensive review", *arXiv preprint arXiv:2311.11883*, 2023, Submitted on 20 Nov 2023. arXiv: 2311.11883 [stat.ML]. [Online]. Available: https://doi.org/10.48550/arXiv.2311.11883.

[38] E. A. Consortium, "Advances and trends in edge ai: A survey", *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6904–6928, 2020.

[39] R. Singh and S. S. Gill, "Edge ai: A survey", *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, 2023. [Online]. Available: https://www.keaipublishing.com/en/journals/internet-of-things-and-cyber-physical-systems.

[40] T. Sipola, J. Alatalo, T. Kokkonen, and M. Rantonen, "Artificial intelligence in the iot era: A review of edge ai hardware and software", *Proceedings of the 31st Conference of FRUCT Association*, 2024.

[41] V. Tsoukas, E. Boumpa, G. Giannakas, and A. Kakarountas, "A review of machine learning and tinyml in healthcare", in *25th Pan-Hellenic Conference on Informatics (PCI 2021)*, ACM, 2021, pp. 69–73. DOI: 10.1145/3503823.3503836.

[42] D. Lara and M. A. Labrador, "A survey on human activity recognition using wearable sensors", *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1192–1209, 2013.

[43] A. De, K. P. Bhatia, J. Volkmann, R. Peach, and S. R. Schreglmann, "Machine learning in tremor analysis: Critique and directions", *Movement Disorders*, vol. 38, no. 5, pp. 717–728, 2023. DOI: 10.1002/mds.29376.

[44] J. Barth, J. Klucken, P. Kugler, *et al.*, "Biometric and mobile gait analysis for early diagnosis and therapy monitoring in parkinson's disease", in *33rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Boston, Massachusetts, USA: IEEE, 2011, pp. 868–871. DOI: 10.1109/IEMBS.2011.6090166.

[45] B. M. Eskofier, S. I. Lee, J.-F. Daneault, *et al.*, "Recent machine learning advancements in sensor-based mobility analysis: Deep learning for parkinson's disease assessment", in *Proceedings of the 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, IEEE, 2016, pp. 655–658. DOI: 10.1109/EMBC.2016.7590796. [Online]. Available: https://doi.org/10.1109/EMBC.2016.7590796.

[46] L. Sigcha, I. Pavón, N. Costa, *et al.*, "Automatic resting tremor assessment in parkinson's disease using smartwatches and multitask convolutional neural networks", *Sensors*, vol. 21, p. 291, 2021. DOI: 10.3390/s21010291.

[47] R. San-Segundo, A. Zhang, A. Cebulla, *et al.*, "Parkinson's disease tremor detection in the wild using wearable accelerometers", *Sensors (Basel)*, vol. 20, no. 20, p. 5817, 2020. DOI: 10.3390/s20205817.

[48] A. Papadopoulos, K. Kyritsis, L. Klingelhoefer, S. Bostanjopoulou, K. R. Chaudhuri, and A. Delopoulos, "Detecting parkinsonian tremor from imu data collected in-the-wild using deep multiple-instance learning", *IEEE Journal of Biomedical and Health Informatics*, vol. 24, no. 9, pp. 2559–2569, 2020. DOI: 10.1109/JBHI.2019.2961748.

[49] A. J. Zhang, "Personalized and weakly supervised learning for parkinson's disease symptom detection", PhD thesis, Carnegie Mellon University, 2019.

[50] W. Roth, G. Schindler, B. Klein, *et al.*, "Resource-efficient neural networks for embedded systems", *arXiv preprint arXiv:2001.03048v3*, 2024. [Online]. Available: https://arxiv.org/abs/2001.03048v3.

[51] H. Han and J. Siebert, "Tinyml: A systematic review and synthesis of existing research", *IEEE International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pp. 269–274, 2023. DOI: 10.1109/ICAIIC54071.2022.9722636.

[52] S. Patel, K. Lorincz, R. Hughes, *et al.*, "Monitoring motor fluctuations in patients with parkinson's disease using wearable sensors", *IEEE Transactions on Information Technology in Biomedicine*, vol. 13, pp. 864–873, 2009. DOI: 10.1109/TITB.2009.2033471. [Online]. Available: https://doi.org/10.1109/TITB.2009.2033471.

[53] G. Rigas, A. Tzallas, M. Tsipouras, *et al.*, "Assessment of tremor activity in the parkinson's disease using a set of wearable sensors", *IEEE Transactions on Information Technology in Biomedicine*, vol. 16, pp. 478–487, 2012. DOI: 10.1109/TITB.2011.2177662. [Online]. Available: https://doi.org/10.1109/TITB.2011.2177662.

[54] S. Roy, B. Cole, L. Gilmore, *et al.*, "High-resolution tracking of motor disorders in parkinson's disease during unconstrained activity", *Movement Disorders*, vol. 28, pp. 1080–1087, 2013. DOI: 10.1002/mds.25399. [Online]. Available: https://doi.org/10.1002/mds.25399.

[55] C. Ahlrichs and A. Samà Monsonís, "Is "frequency distribution" enough to detect tremor in pd patients using a wrist worn accelerometer?", in *Proceedings of the 8th International Conference on Pervasive Computing Technologies for Healthcare*, Oldenburg, Germany, 2014, pp. 65–71.

[56] N. Kostikis, D. Hristu-Varsakelis, M. Arnaoutoglou, and C. Kotsavasiloglou, "A smartphone-based tool for assessing parkinsonian hand tremor", *IEEE Journal of Biomedical and Health Informatics*, vol. 19, pp. 1835–1842, 2015. DOI: 10.1109/JBHI.2015.2462725. [Online]. Available: https://doi.org/10.1109/JBHI.2015.2462725.

[57] M. Braybrook, S. O'Connor, P. Churchward, T. Perera, P. Farzanehfar, and M. Horne, "An ambulatory tremor score for parkinson's disease", *Journal of Parkinson's Disease*, vol. 6, pp. 723–731, 2016. DOI: 10.3233/JPD-160891. [Online]. Available: https://doi.org/10.3233/JPD-160891.

[58] H. Jeon, W. Lee, H. Park, *et al.*, "Automatic classification of tremor severity in parkinson's disease using a wearable device", *Sensors*, vol. 17, p. 2067, 2017. DOI: 10.3390/s17092067. [Online]. Available: https://doi.org/10.3390/s17092067.

[59] H. Kim, W. Lee, A. Kim, *et al.*, "Wrist sensor-based tremor severity quantification in parkinson's disease using convolutional neural network", *Computers in Biology and Medicine*, vol. 95, pp. 140–146, 2018. DOI: 10.1016/j.compbiomed.2018.02.003. [Online]. Available: https://doi.org/10.1016/j.compbiomed.2018.02.003.

[60] M. Hssayeni, J. Jimenez-Shahed, M. Burack, and B. Ghoraani, "Wearable sensors for estimation of parkinsonian tremor severity during free body movements", *Sensors*, vol. 19, p. 4215, 2019. DOI: 10.3390/s19194215. [Online]. Available: https://doi.org/10.3390/s19194215.

[61] P. Pierleoni, A. Belli, O. Bazgir, L. Maurizi, M. Paniccia, and L. Palma, "A smart inertial system for 24h monitoring and classification of tremor and freezing of gait in parkinson's disease", *IEEE Sensors Journal*, vol. 19, pp. 11 612–11 623, 2019. DOI: 10.1109/JSEN.2019.2937101. [Online]. Available: https://doi.org/10.1109/JSEN.2019.2937101.

[62] N. Mahadevan, C. Demanuele, H. Zhang, *et al.*, "Development of digital biomarkers for resting tremor and bradykinesia using a wrist-worn wearable device", *NPJ Digital Medicine*, vol. 3, p. 5, 2020. DOI: 10.1038/s41746-019-0204-9. [Online]. Available: https://doi.org/10.1038/s41746-019-0204-9.