Convolutional Neural Network Based System for Heartbeat Arrhythmia Detection

Ethan Patrick Lynam
21509509

Final Year Project – 2025
B.Sc. in Computer Science and Software Engineering

Department of Computer Science
Maynooth University
Maynooth, Co. Kildare
Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. in Computer Science and Software Engineering.

Supervisor: **Prof. Bryan Hennelly**

Contents

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of B.Sc. Computer Science and Software Engineering qualification, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: [illegible]   Date: 16/3/25

Acknowledgements

I would like to sincerely thank my supervisor Bryan Hennelly for their invaluable guidance and support throughout the development of this project. I am also grateful to John Malaco for their assistance with soldering the hardware, which was crucial to its success.

Abstract

Electrocardiograms are vital for diagnosing heart arrhythmias, but manual interpretation is time-consuming and prone to errors. This project addresses these limitations by developing an automated, real-time system for arrhythmia detection using a 2D convolutional neural network. The system transforms raw ECG data into grayscale images, leveraging the MIT-BIH database for training and validation. To handle data imbalance, arrhythmia images are augmented, creating four slightly altered renditions of the original. The CNN, built using TensorFlow, is optimized for high accuracy in classifying arrhythmias, achieving 93% validation accuracy. A hardware setup, comprising an Arduino Nano 33 IoT microprocessor and an AD8232 ECG module, acquires and transmits ECG data via a WebSocket connection to the software for real-time processing. The system identifies R-peaks, generates heartbeat images centred around each R-peak, and classifies them using the CNN, notifying users of detected arrhythmias. By combining machine learning techniques with accessible hardware, this project aims to improve the accuracy and accessibility of cardiac monitoring, offering a scalable solution for quick and efficient arrhythmia detection.

Chapter 1: Introduction

1.1 Topic and Project Motivation

An electrocardiogram (ECG) is a non-invasive diagnostic tool that plays a crucial role in assessing the hearts rhythm and electrical activity. It is typically one of the first tests a cardiologist will perform when evaluating a patient for suspected heart conditions. One of the key abnormalities an ECG recording can document is arrythmia, an irregular heartbeat [1]. While most arrythmia are generally harmless when experienced occasionally, frequent and prolonged instances and/or specific types of arrythmia can be an indicator of underlying cardiovascular problems that have the potential to be life-threatening [2].

For example, premature ventricular contractions (PVCs)—a common and otherwise harmless type of arrhythmia—can escalate into ventricular fibrillation (VF) if they occur persistently [3]. Ventricular fibrillation is an example of a life-threatening arrythmia that is characterized by quivering of the heart's ventricles. If left untreated it will lead to heart failure and sudden cardiac death [4]. This highlights how important the quick and efficient detection and classification of heart arrythmia from a patient's ECG results is in the field of cardiology.

Despite the critical role ECG analysis plays in identifying and diagnosing arrhythmias, manual interpretation by cardiologists is slow, inefficient and prone to human error. These limitations highlight the need for automated alternatives that can enhance efficiency, accuracy and reliability. With the advancements of machine learning and artificial intelligence in recent years, automated systems adoption is becoming increasingly widespread in the assistance of medical diagnostics, offering faster and more reliable results than traditional methods [5].

However, AI-assisted ECG analysis is still in its early stages. This is due to the complexity and variability of ECG signals, which make it challenging to develop automated systems that can consistently match the expertise of a trained cardiologist [6]. These limitations and difficulties highlight the continued need for improvement in automated ECG analysis. This project attempts to tackle this problem domain by creating an automated, highly accurate system for acquiring ECG data and detecting arrhythmia.

1.2 Problem Statement and Approach

A system needs to be developed that can take a live stream of ECG data from a patient, move that data somewhere suitable for processing, and notify the user of their heartbeat type post-processing. Machine learning methods such as a CNN will be used for the smart and accurate identification of a patient's heartbeat. The CNN will need to be trained to a high standard, emphasizing the need for a large, well-rounded dataset of images. The final system should be able to process, identify and notify of arrhythmia in quick succession with reliable results.

To tackle this problem, the project began with extensive research into existing arrhythmia detection software, focusing on the use of deep learning. Inspired by studies such as "ECG Arrhythmia Classification using a 2-D Convolutional Neural Network" [7] which demonstrated the effectiveness of 2D CNNs for arrhythmia classification, a roadmap was developed to create an automated, real-time ECG monitoring system. The problem was analysed by breaking it into key components that would be dealt with individually, before tying them together at the end: data preprocessing, CNN development, and hardware setup. A prototype was designed using an Arduino Nano 33 IoT and AD8232 ECG module for hardware, paired with a WebSocket for data transmission and processing. The MIT-BIH Arrhythmia Database was used to train and validate a

2D CNN, with a focus on achieving high accuracy. The system was evaluated with a mix of basic software testing and real-world experiments, where controlled variables such as patient-induced noise and elevated heartrate were added to evaluate the systems performance under varying conditions. The experiments were carried out on a handful of consenting patients, to evaluate the model's performance on different patients and see how it behaves with different heart data.

1.3 Metrics

The evaluation of the project will be based on the following metrics:

- **CNN Accuracy**: The proportion of correctly classified heartbeats in both training and validation, and how the architecture is affecting it.
- **Feature Extraction**: The consistency of the R-peak/QRS complex detection.
- **Processing Speed**: The time taken from ECG data acquisition to user notification. This will assess the system's ability to provide real-time predictions.
- **System Architecture:** The evaluation of how and why the system was set-up as is.
- **Analysis:** Investigation of any arrhythmia classifications to identify possible misclassifications and patterns in misclassifications.

1.4 Project

As will be seen throughout the report, the project is split into five key features, and so it can be naturally deduced that the project achieves five significant achievements in its implementation and functionality. These are:

- **Raw ECG data transformation**: Data from the MIT-BIH database is acquired. The R-peaks and their subsequent annotations are found, at which point 128x128 grayscale images are created of each beat, made up of 256 data points to the left and right of the R-peaks location. They are then sorted into their respective beat type, labelled in their annotation.
- **Arrhythmia image augmentation**: For normal heartbeats, a single 128x128 grayscale image is created. For arrhythmia, five 128x128 grayscale images are created, four of which are slightly altered renditions of the original. They are altered with a mix of stretching the data points and multiplying the data with defined polynomials, creating slight differences.
- **2D Convolutional Neural Network**: The CNN's architecture was developed with the TensorFlow library. The data acquired in the previous two achievements was shuffled, downsized and segmented into sets. A Jupyter notebook was subsequently used with these datasets for the training, testing and validation of the CNN, achieving 93% accuracy.
- **Hardware Soldering and Programming**: A microprocessor with Wi-Fi capabilities (Arduino Nano 33 IoT) and an ECG module (AD8232) were acquired, soldered and connected to one another with female-female jumper wires. The microprocessor was subsequently programmed to send the received ECG data with its Wi-Fi module.
- **WebSocket Connection and User Notification**: The microcontroller was setup to act as the WebSocket client, sending ECG data through its Wi-Fi module to a locally run WebSocket server. Once connected and receiving data the server would identify R-peaks, create temporary heartbeat images, pass them to the CNN, and notify the user of a detected arrhythmia through the software terminal. This is done in batches of approximately ten seconds to allow for data acquisition and a processing time window to prevent data loss.

Chapter 2: Technical Background

2.1 Topic material

In recent years, deep learning has revolutionized ECG arrhythmia classification, with 2D convolutional neural networks (CNNs) emerging as a powerful tool. One study [7] transformed 1D ECG signals into 2D grayscale images and used a CNN for classification, achieving 97.53% accuracy on the MIT-BIH Arrhythmia Database. Their approach eliminated the need for manual feature extraction, a common necessity in older studies conducted on deep-learning arrhythmia detection. Their study showcased the potential of 2D representations for capturing spatial patterns in ECG data. The project was heavily inspired by this study.

Building on this, [8] pushed the boundaries further by combining CNNs with LSTM networks to capture both spatial and temporal features. Their hybrid model achieved an impressive 99.21% accuracy on the same dataset, setting a new benchmark for arrhythmia classification. While their method was computationally intensive, it highlighted the importance of leveraging both spatial and temporal information for performance.

2.2 Technical material

2.2.1 MIT-BIH Database Research

The MIT-BIH database is a freely available database of ECG data. It contains 48 records that last for just over 30 minutes, with 47 different patients used for these records. The first half of patients are intended to serve as a representative for the variety of waveforms an arrythmia detector may encounter in routine clinical use. The second half of patients were chosen to include complex ventricular, junctional, and supraventricular arrhythmias and conduction abnormalities [9]. Each records content is stored in the database with the following files:

- **.dat**: binary file with digitized samples, including ECG data.
- **.hea**: header file to describe the content of the signal.
- **.atr:** annotation file to describe the features of the signal.
- **.xws**: for specialised research purposes.

The dat files contain two ECG signals that are associated to an upper and lower signal. The upper signal used is a modified lead II (MLII), while the lower signal used is a modified lead V1, bar a few exceptions. The upper is the preferable signal as the QRS complexes are typically more prominent and therefore more easily detected [9]. See Appendix 1 Tables for annotations found in atr file.

2.2.2 GPU usage for CNN training (CUDA and cuDNN Configuration)

Training the Convolutional Neural Network for the project required significant computational power, and creating an optimal model through trial and error was extremely time consuming. This necessitated the use of a GPU to accelerate the process, as use of the CPU is considerably slower. Since the environment was set up on Windows, additional steps were required to enable GPU acceleration, including installing NVIDIA drivers, configuring CUDA 11.2 and cuDNN 8.1 for WSL, and setting up a CNN-training specific virtual environment with Python 3.10 and TensorFlow 2.10. The need for these specific versions is due to how notoriously finnicky NVIDIA setups are, as only specific versions work with one another [10] [11].

2.2.3 Arduino IDE and Library Research

Programming in the Arduino .ino file structure, with its void setup() and void loop() functions, presented a significant learning curve. The framework's requirement to place initialization code in void setup() and continuous execution logic in void loop() demanded a shift in approach to avoid blocking code and ensure efficient operation. Additionally, integrating libraries for Wi-Fi and WebSocket functionality proved challenging due to the limited selection of compatible libraries. Many libraries were either incompatible with the Arduino Nano 33 IoT or did not work as expected, requiring extensive research to identify suitable options. The WiFiNINA library, used for secure Wi-Fi connections, and the WebSocketsClient library, for real-time communication, required studying documentation and adapting example code to meet the project's requirements. This process showed the challenges of working within the Arduino framework while leveraging unfamiliar libraries.

Chapter 3: The Problem

3.1 Problem Analysis

The technical challenge addressed in this project is the development of an automated, real-time system that can classify eight different heartbeat types from a continuous stream of electrocardiogram data with high accuracy. This involves several key features that must work in conjunction with one another. They are as follows:

- **Data Preprocessing:** A large amount of raw ECG data must be acquired from a freely available source. Each heartbeat needs to be identified and converted into an image suitable for input into a convolutional neural network. A lack of data on specific arrythmia due to their rarity/severity means augmenting existing arrythmia data. This crafting of artificial data will ensure a balanced and robust dataset is acquired.
- **CNN development and optimisation:** A convolutional neural network architecture must be developed and optimised for highly accurate classification between normal and arrhythmic beats. This involves splitting, shuffling and downsizing the pre-processed dataset, to ensure a well-balanced mix of multiple different patients' ECG data is being used for training, testing and validation of the CNN.
- **Hardware Integration:** A basic hardware system must be constructed that can collect ECG data with electrodes. This system must reliably acquire the data and prepare it for transmission back to the software side for processing.
- **Data Transmission:** A stable, fast communication method between the hardware and software must be established. This will act as the bridge between the software responsible for data processing and the hardware responsible for data acquisition.
- **Prediction and Notification System:** The system must receive a stream of live ECG data and re-process each heartbeat into an image. The images must be passed to the now trained and tested CNN for classification. On the event of arrhythmia detection, the user must somehow be notified as soon as possible.

The final system must achieve a hardware set-up that collects ECG data from a patient and passes it to a software set-up. This set-up must have the capability of heartbeat detection and highly accurate arrhythmia detection/classification, while maintaining real-time or near real-time performance, ensuring timely and actionable insights for the use

3.2 Problem UML documentation

See Appendix 2.1 Use Case Diagram for a diagram that has been created in the context of the problem, showcasing how the system will work with external actors as well as the include/extend relationships between the different use-cases.

See Appendix 2.2 Activity Diagram for a diagram that has been created in the context of the problem, showcasing a detailed visual representation of the process/flow within the proposed system.

Chapter 4: The Solution

4.1 Architectural Level

The data collection hardware and classification/notification software are the two major parts of the project's architecture, connected only by their WebSocket client/server relationship. It is a simple set-up as most of the project is undeployed backend functionality.

4.2 High Level

The software is implemented using multiple Python based folders and modules that work in synergy with each other. Inside the repository, they are split up into three main subsections that all carry out specialised tasks for the project. They are called 'preprocessing', 'cnn' and 'ecg_monitoring_server'. The setup varies slightly between each folder, but they all contain a scripts folder for helper code, a data folder for input and/or output data, and a main file where the parent folders' primary function is executed. Outside of this, the hardware setup contains a single .ino (Arduino microprocessor file, identical to C++) file that is uploaded to the Arduino.

4.2.1 Preprocessing Folder

The preprocessing folder is responsible for the acquisition and changing of the data found within the MIT-BIH database, which is stored locally inside the preprocessing folders' 'data' folder. The folders' 'generate_images.py' file carries out this task, and outputs all created images to the 'data' folder. The 'scripts' folder contains four extra python files that all carry additional helper tasks. SeeAppendix 4.1 Preprocessing Folder .

4.2.2 CNN Folder

The CNN folder is responsible for the preparation of data, CNN creation and its subsequent training, testing and validation. It has a folder called 'models' which contains the CNN model's architecture file to be used, and a .keras version of the model that is generated after it has been trained for later classification use. It has a single python file inside the scripts folder which is responsible for prepping the image dataset into training, testing and validation sets. See Appendix 4.2 CNN Folder

4.2.3 Server and Notification Folder

The Server Folder is responsible for the WebSocket server that receives ECG data from the Arduino. Once received, it acts similarly to the preprocessing folder by removing excess noise and performing baseline wander correction before detecting heartbeats in a queue of data. It then creates images around these beats and passes it to the CNN, which outputs its classifications to the terminal. This folder contains the 'client' folder which holds the C++ file responsible for the Arduino nano 33 IoT's functionality. It is stored here purely for viewing purposes and is accompanied by a README.md explaining why it is there. See Appendix 4.3 ECG Server Folder.

4.2.4 Arduino Nano 33 IoT and AD8232

The AD8232 sends data to the Arduino which is loaded with ecg_websocket_client.ino. This file indicates to the Arduino microprocessor what it needs to do. In this case, the file connects to a specified Wi-Fi network, searches and connects to the WebSocket server that is being run over the same Wi-Fi, and sends the received data to the WebSocket server.

4.3 Low Level: Preprocessing

4.3.1 generate_images.py

The main script in data preprocessing is generate_images.py. It is responsible for processing ECG data from the MIT-BIH Database and converting it into 128x128 grayscale images centred around the R-peak. The process_patients_records() function contained in the file follows a pipeline that performs the following:

Data Prep: This is the beginning of the function. It creates two variables 'record_data' and 'annotation' which use the wfdb library to process the dat and atr files associated with the current patient's number. The dat files return tuples with different information, so it is then ensured that the information mapped to the variable is the raw ecg data. It uses a specific lead type from the ecg data, as the ecg data contains two leads and one lead contains more readable data than the other. Finally, the ecg data is passed to three imported functions; denoise_signal, remove_baseline_wander, and detect_rpeaks. See Appendix 5.3 generate_images.py lines 45-64.

Noisy Period Filtering: After data prep, data that is too noisy within the record is identified and flagged. It loops through all annotations and checks for '~', which indicates a change in signal quality. It then checks the subtype field, where 1,2 and 3 indicate the beginning of noisy periods, 0 indicates the end of noisy periods, and -1 indicates other changes we are not interested in. Whenever a noisy period begins, the script records the start index, waits for subtype 0 indicating clean data, and marks this index as the end index. The script then moves onto looping through each detected R-peak, and will skip any R-peaks that fall between the documented noisy periods. See Appendix 5.4 generate_images.py lines 68-111 .

Image Creation: Within the R-peaks indices loop, any non-noisy periods come to the image creation section of the script. It begins by calculating the absolute differences between the current R-peak index and all annotation samples. The annotation sample with the smallest difference is marked as the current R-peak/heartbeats annotation. This sample is then passed to the annotation map created at the start of the file, which associates the annotation symbol to its full name (symbol N = Normal, symbol / = Paced Beat, etc.). If the symbol isn't in the dictionary it defaults to OTHER, as not all beats/annotations in the MIT-BIH are covered by the project. After this, it defines the range around the R-peak which is 256 data points to the right and left of the beat. Using the matplotlib library, a plot is created of the defined range. This plot is then saved as a grayscale image of size 128x128 within the folder data/created_images/{heartbeat type}. Finally, the imported function beat_augment is used to create a further four images of any heartbeats that are not normal (arrhythmia). See Appendix 5.5 generate_images.py lines 114-144.

Multiprocessing: The actual functionality of the file is contained within a function purely so that the 'os' and 'multiprocessing' libraries can utilise it for processing multiple records at the same time. This was integrated later in development as the file's execution speed was too slow. See main where function is called with multiprocessing: Appendix 5.2 generate_images.py lines 147-150.

There are eight different heartbeat types that preprocessing is responsible for acquiring and creating images of. There are more within the MIT-BIH, however this project focuses on these specific eight. See Appendix 4.10 Heartbeat Types Covered in this Project - Preprocessing Generated Images Examples of Each.

4.3.2 augment_images.py

The function beat_augment() in this file performs four different augmentation combinations on an inputted arrhythmic beat to generate four new versions of the original. The original arrhythmic beat is taken and stretched with the use of the scipy library, which interpolates the beat to 1.25x its original length with the 'interp1d' function. This simulates the heartbeats at a slightly different speed to help introduce variation. The original and stretched beats are then multiplied by their respective polynomials:

Original Polynomials:

$$P_{pos}(x) = 0.3x^2 + 0.7x + 0.2$$

$$P_{neg}(x) = -0.2x^2 - 0.5x + 0.8$$

Stretched Polynomials:

$$P_{pos,stretched}(x) = 0.4x^2 + 0.9x + 0.3$$

$$P_{neg,stretched}(x) = -0.3x^2 - 0.6x + 0.7$$

This creates four new beats, as the polynomials will either slightly amplify or slightly smooth certain peaks and fluctuations, while still maintaining the heartbeats core shape. See Appendix 4.9 Data Augmentation for an example. See Appendix 5.6 augment_images.py lines 15-52 for the code implementation of this logic.

4.3.3 detect_rpeaks.py

The function detect_rpeaks() in this file is responsible for the detection of heartbeats within a passed ECG signal. This is done using the ecg functionality within the biosppy signals library, which works to identify important features inside a piece of ECG signal, such as R-peaks, P-waves and T-waves. Once identified, the library returns a dictionary containing this important information, and the function returns only the R-peaks. See Appendix 5.7 detect_rpeaks.py.

4.3.4 ecg_baseline_wander.py

The function remove_baseline_wander() in this file is responsible for removing baseline wander from the ECG signal, which refers to low-frequency fluctuations that distort the ECG waveform. To achieve this, the function applies a **high-pass** Butterworth filter using the scipy.signal library. The filter is designed with a cutoff frequency of 0.5 Hz, which removes slow-moving baseline variations while preserving the higher-frequency components of the ECG signal. This process helps to "straighten" the ECG data without distorting its overall shape, making it more suitable for accurate analysis. See Appendix 5.8 ecg_baseline_wander.py .

4.3.4 ecg_denoise.py

The function denoise_signal() in this file is responsible for removing noise from an ECG signal by applying wavelet decomposition. The signal is first decomposed into multiple sub-signals using a discrete wavelet transform with the 'db5' wavelet and a decomposition level of 10. The function then eliminates the low-frequency and high-frequency components by zeroing out the corresponding wavelet coefficients, keeping only the middle frequencies that contain the main

ECG signal features. Afterward, the inverse wavelet transform is applied to reconstruct the denoised ECG signal, effectively removing unwanted noise while preserving the important characteristics of the ECG waveform. See Appendix 5.9 ecg_denoise.py .

4.4 Low Level: CNN

4.4.1 CNN Architecture

The CNN is the core of the classification system, responsible for analysing the ECG images and identifying arrhythmias. The model is created with TensorFlow inside of the arrhythmia_detection_cnn.py file. See Appendix 2.4 CNN Architecture Diagram and Appendix 4.5 CNN Code Architecture for an overview of the CNN's implemented architecture.

Regularization: An L2 Regularization of 0.001 was added to the convolutional layers and first dense layer to penalise large weights and encourage simpler patterns and combat overfitting.

Dropout: A small amount of dropout was implemented to remove a very small fraction of the neurons during training. This prevented the model from relying too heavily on specific neurons.

Optimizer: The Adam optimiser was used with a modified learning rate of 0.0002. It was slowed down to help it generalise better and avoid any overfitting.

The model was trained over 15 epochs and consistently gets 97% training accuracy and 93% validation accuracy. Unfortunately, the model experiences very slight validation overfitting, hence the regularization and slowed learning rate being implemented to combat this as best it can. See Appendix 4.4 CNN Training Epochs.

4.4.3 dataset_prep.py

The dataset_prep.py script is used within the Jupyter notebook to prepare the dataset of images created by preprocessing. It organises the images into structured folders of training, testing and validation, ensuring a balanced distribution of data across the eight different classes and patients. It also down samples larger classes to avoid class imbalances. It carries out these functionalities to ensure the CNN is using well balanced datasets for proper learning and optimisation. See Appendix 5.16 dataset_prep.py for its implementation.

- Creates three main folder data/training, data/validation and data/testing to store the training, validation and test data, then iterates through created images. Skips OTHER as only interested in 8 arrhythmia types for project.
- Shuffles the images within each class folder to ensure the three CNN sets have a balanced mix of random patients' data.
- Limits the number of images within each class to 3000 to address class imbalances.
- Finally, loops through the files in each class set, creates symbolic links to the original images, and organizes them into training, validation, and test folders. This ensures efficient storage usage while maintaining a balanced dataset structure.

4.5 Low Level: Server and User Notification

4.5.1 server.py

The server.py script is the core component of the server and user notification system. It sets up a WebSocket server to receive ECG data from connected clients, processes the data, and classifies heartbeats with the trained CNN. The script is designed to handle real-time ECG data streaming, ensuring the data is processed without significant loss. It is made up of three major components listed below:

main(): The main function is the entry point for the script. It initializes and starts the WebSocket server with the WebSocket's library, which listens for incoming connections from clients. See Appendix 5.10 server.py: main() for code implementation.

- Initializes server to 0.0.0.0 (all available network interfaces) and listens on port 9000, making it accessible to clients on the same network and port.
- The server runs indefinitely with "await server.wait_closed()", which requires manual termination.

handle_connection(): The handle connection function manages the WebSocket connection to a client and executes functionality when set requirements have been met. See Appendix 5.12 server.py: handle_connection() for code implementation.

- Receives the ECG data from the client, which is coming as a continuous stream, converts each message (data-point) to an int, adds the data to a shared queue taken from the collections library, and adds to a shared counter which is tracking how many messages have been received.
- The counter is tracking to wait until 1625 messages/data-points have been received. This specific number is used as 1624 is the minimum number of data points required for signal processing by the scipy ecg library.
- Once the counter and therefore the queue reach 1625 data point capacity, and the shared processing flag is False, a new thread is created with the threading library. This thread then executes the process_data() function, and the processing flag is set to True.
- A secondary thread for data processing was decided upon because without it, data loss would occur as the program is caught up on processing and not receiving the data stream.

process_data(): The process data function is responsible for processing the ECG data currently stored in the shared queue. It performs a series of signal processing steps with imported helper functions to prepare the data for heartbeat classification. See Appendix 5.11 server.py: process_data().

- Makes the shared processing flag equalled to true, preventing the system from starting another processing Thread while the previous queue has not been completed. This should not happen and is a precautionary step.
- Converts the ECG data queue to a NumPy array for processing which is then passed to three imported helper functions in quick succession; denoise_signal(), remove_baseline_wander(), and detect_rpeaks().
- Uses the detected R-peaks in an imported helper function, create_images(), which creates images around the R-peaks with matplotlib library similarly to the preprocessing section.
- It then loops through the created images, classifies them with another imported helper function classify_heartbeat(), and deletes the images to save room for the next batch.
- Finally, the shared counter is reset to 0, the ECG queue is cleared, and the processing flag is set to False. It forces 'garbage collection' with the gc library to prevent errors caused by using matplotlib in secondary threads.

4.5.2 heartbeat_classification.py

The function classify_heartbeat() found in this file is responsible for classifying a heartbeat image using a pre-trained CNN model. See Appendix 5.13 heartbeat classification.py for code implementation.

- Loads the pretrained CNN model from its previously created keras file with the tensorflow library and defines a 'class_labels' list with each heartbeat types full name matching to the CNN's heartbeat type classification arrays output.
- The function is defined and the input image is loaded with the TensorFlow library.
- The image is converted to a NumPy array and expanded to include a batch dimension, as the model expects an input that includes a batch size.
- The image is then passed to the CNN which outputs a list of probabilities for each type. The type with the highest probability is selected, and the full name is retrieved from the corresponding class_labels list to be outputted to the terminal.

4.6 Low Level: Hardware

4.6.1 ecg_websocket_client.ino

This file implements the hardware sides functionality for capturing ECG data and transmitting it to a WebSocket server. It runs on a microcontroller (Arduino Nano 33 IoT) and is dependent on an external ECG sensor for its data (AD8232).

- Void setup(): initializes the serial monitor which allows outputting to a terminal in the IDE. Connects the microcontroller to the Wi-Fi with predefined SSID and Password variables, using the WiFiNINA library. Connects to the WebSocket server using predefined server address and server port variables, using the WebSocketsClient library. See Appendix 5.14 ecg websocket client.ino: void setup() for code implementation.
- Void loop(): Continuously captures data from analog pin A0, which has been wired to the output of the AD8232. It converts each datapoint to a string for transmission which is a requirement of the WebSocketsClient library. Then, it sends the data to the server using the webSocket.sendTXT() function. Finally, it introduces a delay to mimic a sampling rate. See Appendix 5.15 ecg websocket client.ino: void loop() for code implementation.
- webSocketEvent(): This is taken from the WebSocketsClient libraries GitHub page. It was necessary to implement proper WebSocket functionality [12].

4.6.2 Hardware Schematics

As previously mentioned, the Arduino Nano 33 IoT microcontroller and AD8232 ECG sensor were used as the hardware components for ECG acquisition and sending. Each component has a header strip attached to its pins, and female-female jumper wires were attached to the header strips male ends for connection. See Appendix 5.1 Hardware Wiring Diagram.

- 3.3V: power supply.
- GND: ground to complete electrical circuit.
- OUTPUT: connects to an analog pin to send the acquired data.
- LO- & LO+: used for lead-off detection. Decided not to integrate utilisation.

The AD8232 requires three different electrode lead types to be tracking data simultaneously. These are RA, LA and RL and their placement on the body is crucial for proper data acquisition. See Appendix 5.2 AD8232 Electrode Placement.

4.7 Implementation

During implementation, many unexpected challenges presented themselves. The successful acquisition of data from the MIT-BIH was a source of many problems. The data files that contain the ECG data have two electrode leads, as previously discussed, but I was not aware of this during initial development. I was automatically using the top leads with record_data[0], but patients 114 and 207 have their top and bottom leads switched. Due to this, these patients' data was extremely unclear and messy. After further investigation I discovered the problem, but it caused me great confusion and is why this code is present in generate_images.py: Appendix 5.17 Code Snippet 1 .

Another problem in the acquisition of ECG data from the MIT-BIH was the detection and avoidance of noisy periods in the data. I had initially been processing all beats, but noticed large sections of very noisy, unreadable images being created. Obviously, these would be detrimental to the CNN's learning and evaluation, so I had to fix it. I discovered the start and end of noisy periods are annotated by a '~' and so added to the script to skip everything after '~' until another is detected. However, this would not work either as '~' denoted cc, nn, nc and cn (see Appendix 1.3 Signal annotations in the MIT-BIH Database for meaning). I had to remake it so it would now look for subtypes, where cc = 0, nn = 3 etc. but this did not work either as unrelated things were picked up as = -1. Finally, I combined the two methods, where '~' AND subtype != 0 would cause all data to be skipped until '~' AND subtype = 0 was encountered again. Appendix 5.18 Code Snippet 2.

After writing a simple script that takes from the images created from the MIT-BIH database and segments them into training, testing and validation folders, I quickly realized three things.

- Some arrhythmia types have under one thousand images due to their rarity, while normal heartbeats have almost 70,000 images. This would create massive generalisation in the CNN's learning.
- The images are saved in order due to their names being the patient number followed by the heartbeat number. This would create training, testing and validation segments with a total lack of some patient's data, where multiple data sources are a necessity.
- Copying the images into their new folders inside the CNN folder was extremely inefficient for both time and storage.

This caused me to expand on dataset_prep.py greatly, as I needed to address these problems. The current rendition solves the data problems by shuffling and downsizing the data before segmenting, and the storage problems with the use of symlinks. Appendix 5.19 Code Snippet 3.

The CNN setup was another interesting implementation. Initially the CNN architecture was being based off AlexNet, but I quickly discovered that the likes of these complex CNNs are specialised for massive datasets. I developed a much simpler CNN that was gaining high accuracy very quickly (after just 5 epochs). This eventually became the current CNN rendition, which has added dropout layers, L2 regularisation and an Adam optimiser with a slower learning rate. This allowed for a CNN architecture that retains its simplicity and allows for higher epochs with little overfitting.

Originally the project was using the microprocessor as a means of sending the ECG data to the cloud. A Firebase Realtime database was to receive the data, and all processing was to be done on the cloud. After further development this was switched out for the WebSocket server, as hosting of TensorFlow CNNs on the cloud proved to be difficult. Furthermore, processing in the cloud was an unnecessary addition for the scope of this project.

The AD8232 Heartbeat Module came without the header strip attached. I was faced with ordering a new one or attempting to solder myself. After careful consideration I soldered the male ends of male-female jumper wires to the AD8232 pins. Unfortunately, during testing, the data was very noisy, and I suspected this was the cause. After seeking assistance, the wires were removed, a header strip was soldered on, and female-female jumper wires were used. The data was still coming back noisy, and after further investigation the problem turned out to be in how I was viewing the ECG data being received. It was a phantom roadblock, but at least the hardware wiring was improved with a header pin, creating reusability and connection reliability.

Chapter 5: Results and Evaluation

5.1 Solution Verification

To verify the solution that was implemented, a few parts of the system were tested and checked. The CNN was tested with images from the MIT-BIH that were **not** used in the training or validation sets. It achieved an average accuracy of 93%, showcasing its ability to correctly classify most heartbeats it is passed. Its performance was also assessed through its training and validation loss, both of which displayed significant drops before evening out. These three factors show a healthy CNN that is achieving high accuracy without much overfitting. See Appendix 4.6 CNN Model Accuracy and Appendix 4.7 CNN Model Loss. Additionally, the hardware setup was verified by comparing the ECG data acquired from the AD8232 module with standard ECG waveforms, ensuring that the data collected was consistent with expected data. The images created of the collected data were up to standard, see Appendix 4.8 AD8232 Acquired Data Images.

5.2 Software Design Verification

The software design was verified using the previously made UML interaction diagrams (see Appendix 2.1 Use Case Diagram and Appendix 2.2 Activity Diagram) to model the system's interactions between components. These diagrams were used to validate the preprocessing of training data, acquisition and prep of hardware data, and classification of the acquired data.

5.3 Software Verification

5.3.1 Test approach

I tested the project with a mix of manual testing, sub-system testing and system testing.

- Manual testing: Individual components such as the image generation script and CNN classification were manually tested to ensure expected outcomes.
- Sub-system testing: The interaction between the WebSocket server and client were tested to ensure correct data was being sent with no data loss.
- System testing: The entire systems functionality was tested, ensuring it was working as expected.

5.3.2 Tests

- During manual testing, I manually confirmed whether the outputs of the system were correct. This would have entailed checking outputted images from preprocessing and comparing them to the waveform visualisation of the section that had been processed from the MIT-BIH. For the image output of the hardware, the images were compared to a live waveform visualiser. For CNN classifications, I altered the server.py script to not delete the created images after classification had been performed and compared the images to the results. Through this testing I discovered the CNN tended to classify what looked like normal beats as Right Bundle Branch beats.

- For sub-system testing I performed ping/pong messages initially to confirm the WebSocket's proper connection and later outputted every single data point being sent/received on both sender and receiver terminal. The data points were then compared to validate there was no data loss.
- System testing simply involved running the system and checking each part for any flaws or mistakes. None seemed to be present.

5.3.4 An interpretation of the results

The results demonstrated that the system performs well in all areas. The possible RBB misclassifications could be a potential problem, but I am open to the idea that the CNN could be correct, and I have a recurrent RBB beat! This isn't too far-fetched when you consider the recurrent RBB classification was not present when tested on two other people. Furthermore, The CNN is comparable to existing solutions, such as the study by this study [7], which achieved 97.53% accuracy using a 2D CNN. While the current system's accuracy is slightly lower, it offers the added benefit of real-time operation and integration with accessible hardware. Overall, I believe the high accuracy of the CNN paired with the lack of any serious problems discovered while testing shows the systems reliability for arrhythmia detection.

Chapter 6: Conclusions

6.1 Implications of Work Completed

The development of an automated, real-time ECG monitoring and arrhythmia detection system using a 2D convolutional neural network (CNN) has significant implications for the field of cardiology and healthcare technology. By leveraging machine learning and accessible hardware, this project has attempted to address the limitations of manual ECG interpretation, such as inefficiency, human error, and the time-consuming nature of reviewing large datasets. The system provides a scalable, cheap solution for real-time arrhythmia detection, and under further optimisation could be deployed in clinical settings, remote patient monitoring, or even personal health devices. In its current state, it has the potential to improve early diagnosis of life-threatening arrhythmias, reduce the workload of cardiologists, and enhance patient outcomes through timely intervention.

6.2 Contributions to the State-of-the-Art

Although the use of a 2D CNN for arrhythmia classification was inspired by existing research that performs the same thing, this project attempts to build on it by introducing a streamlined approach for real-time patient processing with an integrated ECG acquisition system. The use of an Arduino Nano 33 IoT and AD8232 demonstrates how advanced ECG analysis and arrhythmia detection systems can be achieved with low-cost, easily accessible components.

6.3 Results Discussion

The results of this project are potentially generalizable due to the use of the MIT-BIH Database, which is regarded as a benchmark for ECG analysis. The dataset's contents encompassing 47 patients with various arrhythmia provides a strong foundation for training and validating the system, particularly with the added diversity that is brought with the augmentation of arrhythmia. Additionally, the current implementation using the Arduino Nano 33 IoT and AD8232 ECG module consistently returns expected normal heartbeat classifications when micro-tested on myself, a 'patient' with no known heart issues.

However, while the MIT-BIH database includes a variety of arrhythmia cases, the real-world performance of the system depends on its ability to handle the variability of ECG signals. ECG recordings can be affected by factors such as sensor placement, signal noise, and heart differences among individuals. To confirm the system's generalisability, it would need to be tested further in real-world environments on patients with known heart problems or arrhythmia, or on external datasets featuring diverse ECG data.

Furthermore, although the project demonstrates solid results, there are multiple potential threats to its validity through its limitations. The CNN used is quite simple in comparison to other models developed for the same purposes, and the MIT-BIH dataset, although quite diverse, does not fully represent the variability of ECG signals in the real world, and therefore does not fully prepare the system for such cases. The setup of the hardware and QRS detection algorithm also introduces risks through its approach, since it is built with publicly available, easily accessed components that may not be fully reliable for such an intricate task that absolutely necessitates high, dependable accuracy.

6.4 Future Work

There are some limitations within this project that could be addressed, such as:

- **Improved CNN Architecture:** Exploration of more advanced CNN architectures that have improved classification accuracy would improve the systems reliability.
- **Real-Time ECG Viewing:** The project took a back-end approach and does not have any form of front-end. An ECG monitor could be added that allows for real-time viewing of heartbeats, which would give a view of any detected arrhythmia rather than just an alert.
- **Remote Monitoring:** Moving the system to a mobile platform, adding a battery to the hardware and utilising the cloud could allow for remote monitoring and viewing, enhancing its ability for timely interventions in emergencies.

6.5 Final Thoughts

This project does a good job of demonstrating the feasibility of using machine learning algorithms in combination with readily available hardware to develop a real-time ECG monitoring and arrhythmia detection system. While the system achieves high accuracy and could be considered reliable, there is plenty of room for improvement in its dependability to identify QRS-complexes, identify and classify arrhythmia, and collect clean, readable ECG data. Addressing these limitations using more sophisticated machine learning algorithms and ECG monitoring systems would boost the projects potential to be an invaluable tool in the field of cardiology.

References

[1] D. S. Desai and S. Hajouli., “Arrhythmias,” National Libray of Medicine, 2023.

[2] H. Methodist, “When should you worry about arrhythmia?,” 2020. [Online]. Available: https://www.houstonmethodist.org/blog/articles/2020/jan/when-should-you-worry-about-arrhythmia.

[3] M. Clinic, “Premature ventricular contractions (PVCs) - Symptoms and causes,” 2022. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/premature-ventricular-contractions/symptoms-causes/syc-20376757.

[4] “Ventricular fibrillation - Symptoms and causes,” Mayo Clinic, 2022. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/ventricular-fibrillation/symptoms-causes/syc-20364523.

[5] M. J. Ferreira, “AI in Healthcare: Revolutionising Diagnosis Speed and Accuracy,” Whitesmith, 2025.

[6] X. L. a. J. W. Y. Liu, “Challenges in AI-Assisted ECG Analysis: Variability and Complexity of Signals,” Journal of Biomedical AI Research, 2024.

[7] T. J. N. H. M. K. D. K. D. K. D. &. K. Y.-H. Jun, “ECG arrhythmia classification using a 2-D convolutional neural network.,” IEEE, 2018.

[8] Ö. P. P. T. R. S. &. A. U. R. Yildirim, “Arrhythmia detection using deep convolutional neural network with long duration ECG signals.,” 2018.

[9] G. B. Moody, “MIT-BIH Arrhythmia Database Directory,” Massachusetts Institute of Technology, Massachusetts, 1997.

[10] N. Developer, “CUDA Toolkit 11.2 Downloads,” NVIDIA, [Online]. Available: https://developer.nvidia.com/cuda-11-2-0-download-archive.

[11] N. Developer, “NVIDIA cuDNN Installation Guide,” NVIDIA, [Online]. Available: https://docs.nvidia.com/deeplearning/cudnn/installation/latest/index.html.

[12] A. Links, “WebSocketsClient [software library],” 2024.

[13] Admin, “Real-Time ECG Monitoring with AD8232 & Arduino | Personal Health Tracking,” unknown. [Online]. Available: https://diyprojectslab.com/ecg-monitoring-with-ad8232-arduino/.

[14] D. Shah, “Understating ECG Sensors and How to Program one to Diagnose Various Medical Condition,” 2021. [Online]. Available: https://circuitdigest.com/microcontroller-projects/understanding-ecg-sensor-and-program-ad8232-ecg-sensor-with-arduino-to-diagnose-various-medical-conditions.

[15] "Electrocardiogram (ECG)," [Online]. Available: https://www2.hse.ie/conditions/electrocardiogram-ecg/#:~:text=An%20electrocardiogram%20(ECG)%20is%20a,see%20if%20they're%20unusual..

Appendices

Appendix 1 Tables

Appendix 1.1 Arrhythmia Annotations of MIT-BIH Database

Symbol	***Meaning***
· *or* N	Normal beat
L	Left bundle branch block beat
R	Right bundle branch block beat
A	Atrial premature beat
a	Aberrated atrial premature beat
J	Nodal (junctional) premature beat
S	Supraventricular premature beat
V	Premature ventricular contraction
F	Fusion of ventricular and normal beat
[	Start of ventricular flutter/fibrillation
!	Ventricular flutter wave
]	End of ventricular flutter/fibrillation
e	Atrial escape beat
j	Nodal (junctional) escape beat
E	Ventricular escape beat
/	Paced beat
f	Fusion of paced and normal beat
x	Non-conducted P-wave (blocked APB)
Q	Unclassifiable beat
|	Isolated QRS-like artifact

Appendix 1.2 Arrhythmia Annotations Used for Image Processing

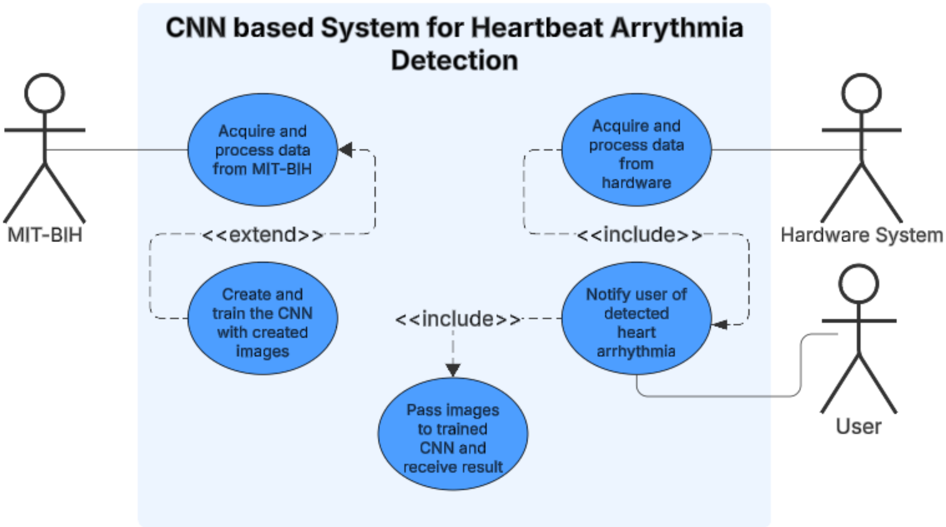
Symbol	***Meaning***
N	Normal beat
L	Left bundle branch block beat
R	Right bundle branch block beat
A	Atrial premature beat
V	Premature ventricular contraction
!	Ventricular flutter wave
E	Ventricular escape beat
/	Paced beat
All Others	Every other arrhythmia type is placed into 'Other' as not interested in training for other arrhythmia

Appendix 1.3 Signal annotations in the MIT-BIH Database

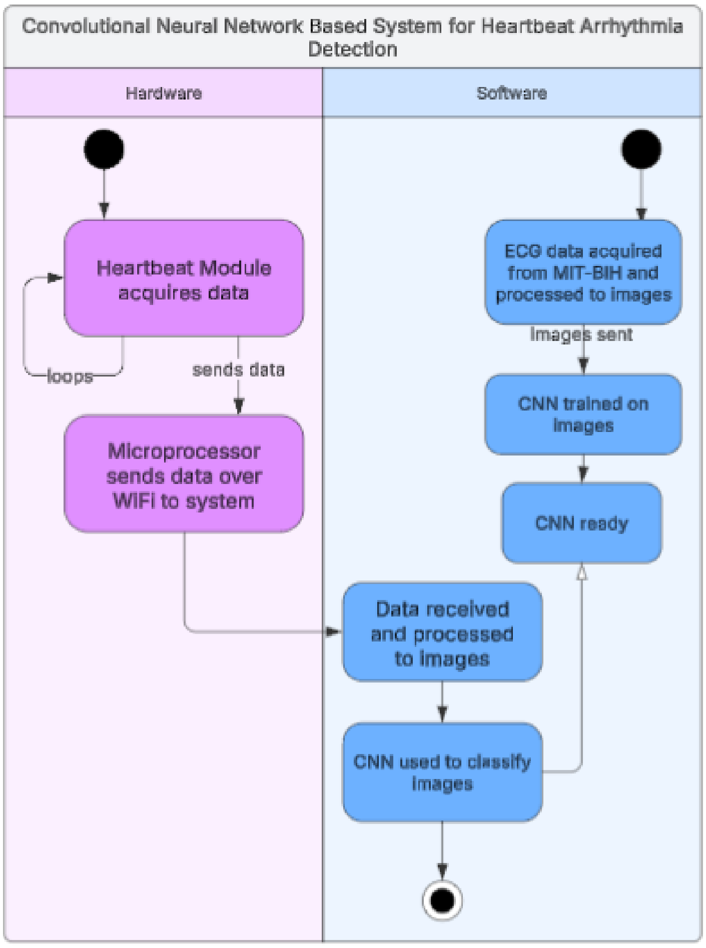
Symbol	Meaning
qq	Signal quality change: the first character (\`c' or \`n') indicates the quality of the upper signal (clean or noisy), and the second character indicates the quality of the lower signal
U	Extreme noise or signal loss in both signals: ECG is unreadable
M (*or* MISSB)	Missed beat
P (*or* PSE)	Pause
T (*or* TS)	Tape slippage

Appendix 2 Diagrams

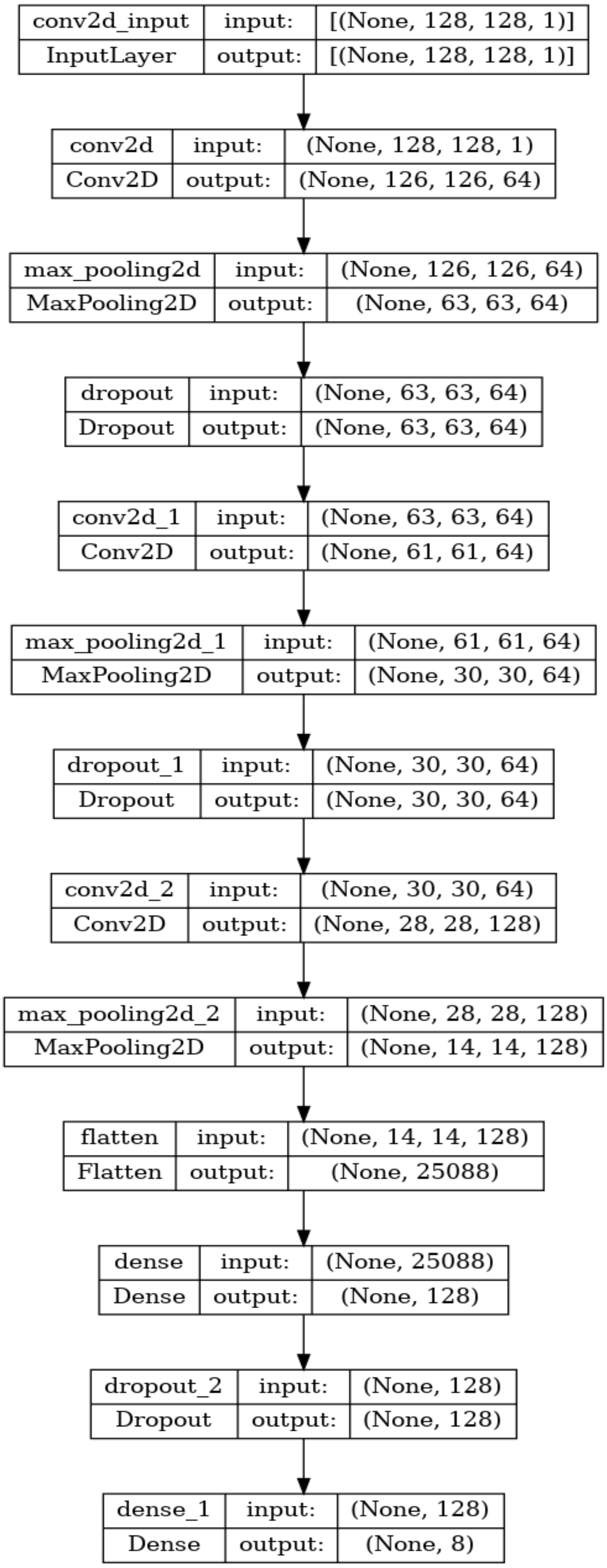
Appendix 2.1 Use Case Diagram



Appendix 2.2 Activity Diagram

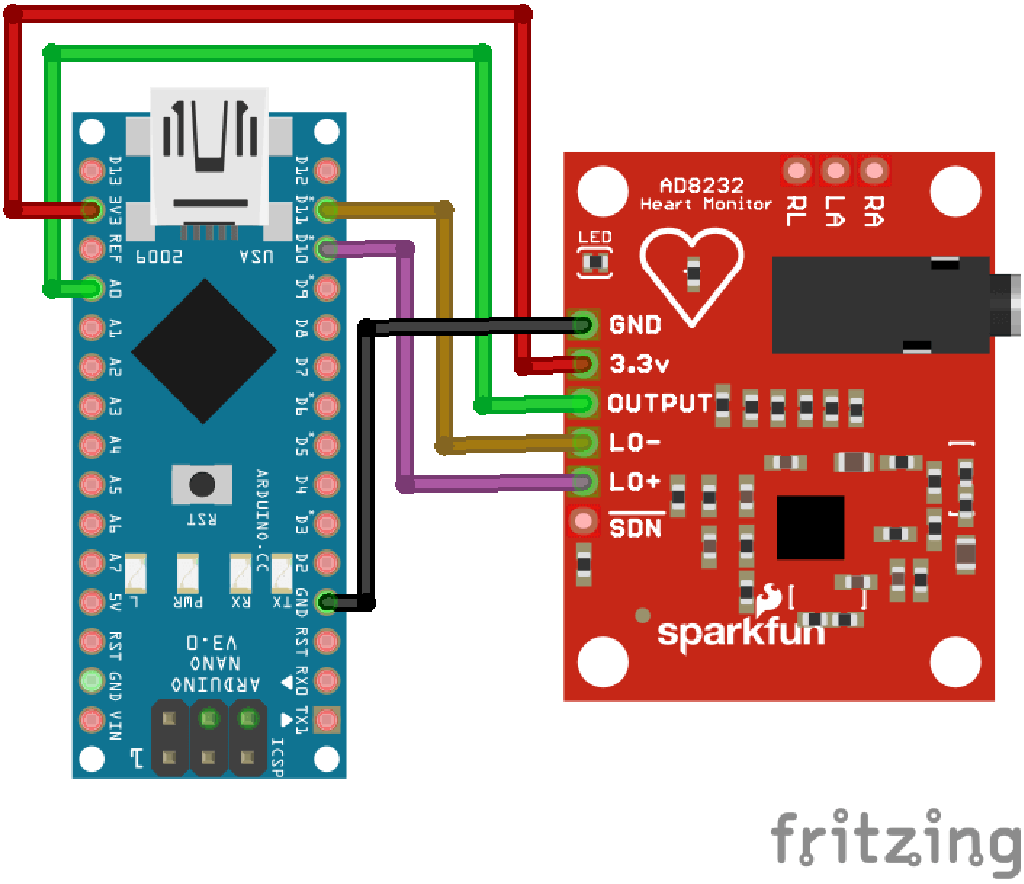


Appendix 2.4 CNN Architecture Diagram



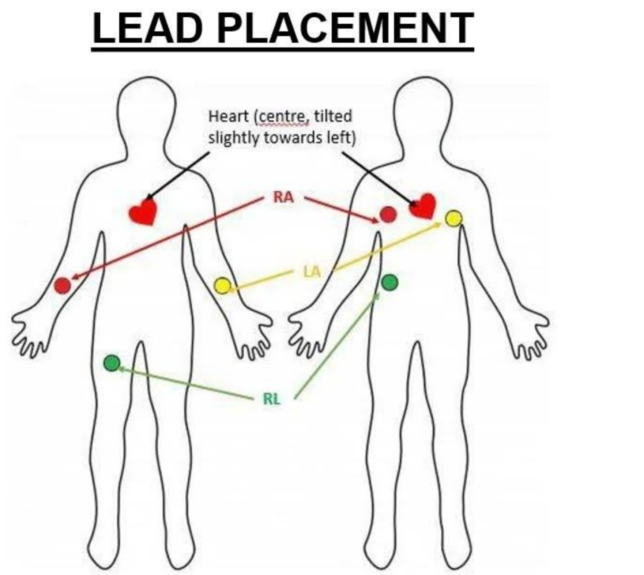
Appendix 3 Hardware Schematics

Appendix 3.1 Hardware Wiring Diagram

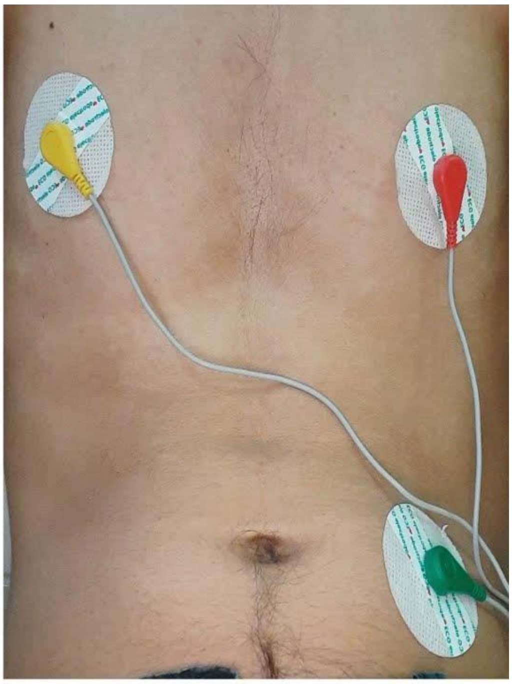


[13]

Appendix 3.2 AD8232 Electrode Placement



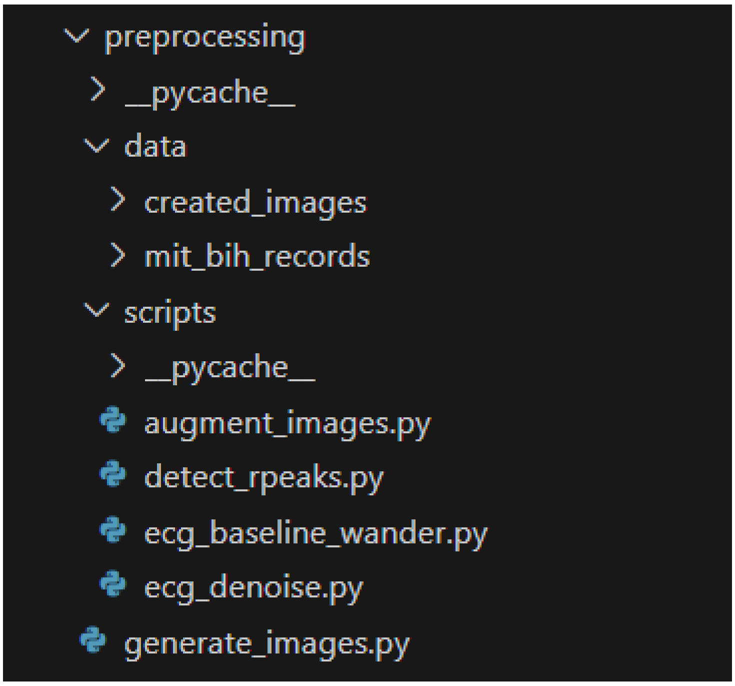
The image on right is a selfie, hence the placement of the electrodes seems to be mirrored.



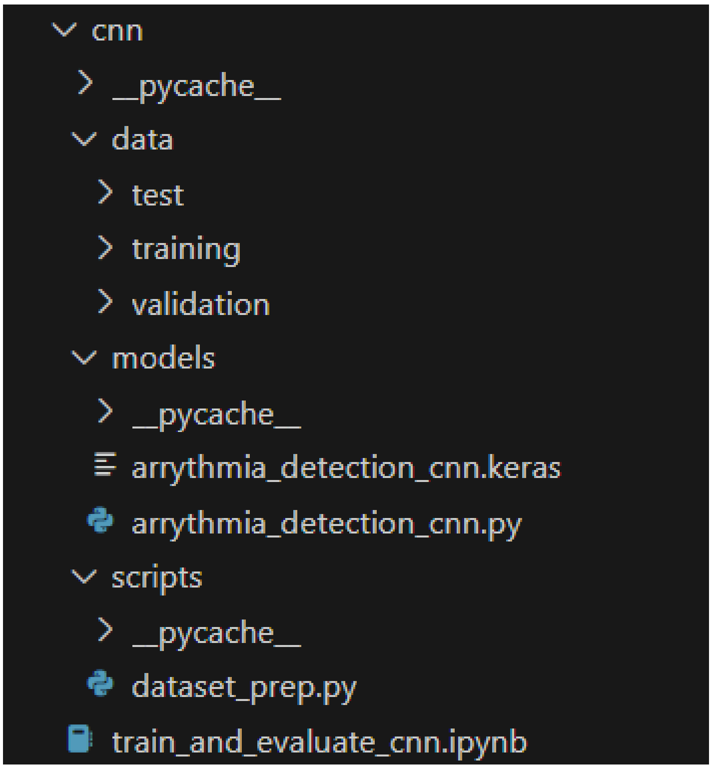
[14]

Appendix 4 Screenshots of Implementation and Results

Appendix 4.1 Preprocessing Folder



Appendix 4.2 CNN Folder



Appendix 4.3 ECG Server Folder

```
ecg_monitoring_server
  > __pycache__
  v client
      ecg_websocket_client.ino
      README.md
  > data
  v scripts
    > __pycache__
      detect_rpeaks.py
      ecg_baseline_wander.py
      ecg_denoise.py
      generate_heartbeat_images.py
      heartbeat_classification.py          2
    server.py                              2
```

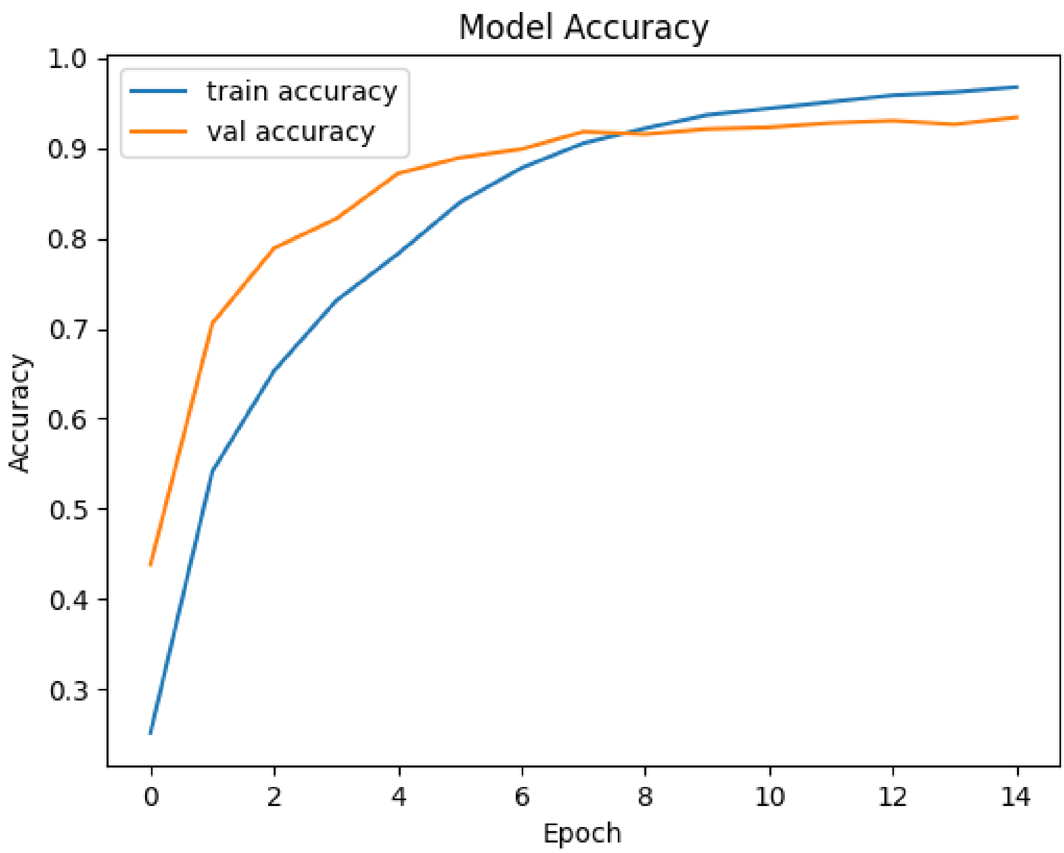
Appendix 4.4 CNN Training Epochs

```
432/432 [==============================] - 22s 33ms/step - loss: 3.3851 - accuracy: 0.2517 - val_loss: 1.5933 - val_accuracy: 0.4381
Epoch 2/15
432/432 [==============================] - 15s 34ms/step - loss: 1.3946 - accuracy: 0.5414 - val_loss: 1.0559 - val_accuracy: 0.7056
Epoch 3/15
432/432 [==============================] - 14s 32ms/step - loss: 1.1120 - accuracy: 0.6531 - val_loss: 0.8736 - val_accuracy: 0.7890
Epoch 4/15
432/432 [==============================] - 13s 29ms/step - loss: 0.9216 - accuracy: 0.7306 - val_loss: 0.7544 - val_accuracy: 0.8215
Epoch 5/15
432/432 [==============================] - 14s 32ms/step - loss: 0.7951 - accuracy: 0.7826 - val_loss: 0.6435 - val_accuracy: 0.8717
Epoch 6/15
432/432 [==============================] - 14s 31ms/step - loss: 0.6639 - accuracy: 0.8396 - val_loss: 0.5834 - val_accuracy: 0.8892
Epoch 7/15
432/432 [==============================] - 15s 33ms/step - loss: 0.5543 - accuracy: 0.8779 - val_loss: 0.5380 - val_accuracy: 0.8988
Epoch 8/15
432/432 [==============================] - 14s 32ms/step - loss: 0.4759 - accuracy: 0.9054 - val_loss: 0.5226 - val_accuracy: 0.9181
Epoch 9/15
432/432 [==============================] - 14s 32ms/step - loss: 0.4297 - accuracy: 0.9218 - val_loss: 0.5286 - val_accuracy: 0.9153
Epoch 10/15
432/432 [==============================] - 14s 33ms/step - loss: 0.3889 - accuracy: 0.9366 - val_loss: 0.5108 - val_accuracy: 0.9211
Epoch 11/15
432/432 [==============================] - 15s 34ms/step - loss: 0.3607 - accuracy: 0.9439 - val_loss: 0.5116 - val_accuracy: 0.9229
Epoch 12/15
432/432 [==============================] - 14s 32ms/step - loss: 0.3320 - accuracy: 0.9511 - val_loss: 0.5258 - val_accuracy: 0.9277
Epoch 13/15
432/432 [==============================] - 15s 34ms/step - loss: 0.3122 - accuracy: 0.9585 - val_loss: 0.5164 - val_accuracy: 0.9303
Epoch 14/15
432/432 [==============================] - 15s 34ms/step - loss: 0.2983 - accuracy: 0.9620 - val_loss: 0.5431 - val_accuracy: 0.9262
Epoch 15/15
432/432 [==============================] - 15s 34ms/step - loss: 0.2729 - accuracy: 0.9676 - val_loss: 0.4923 - val_accuracy: 0.9341
```

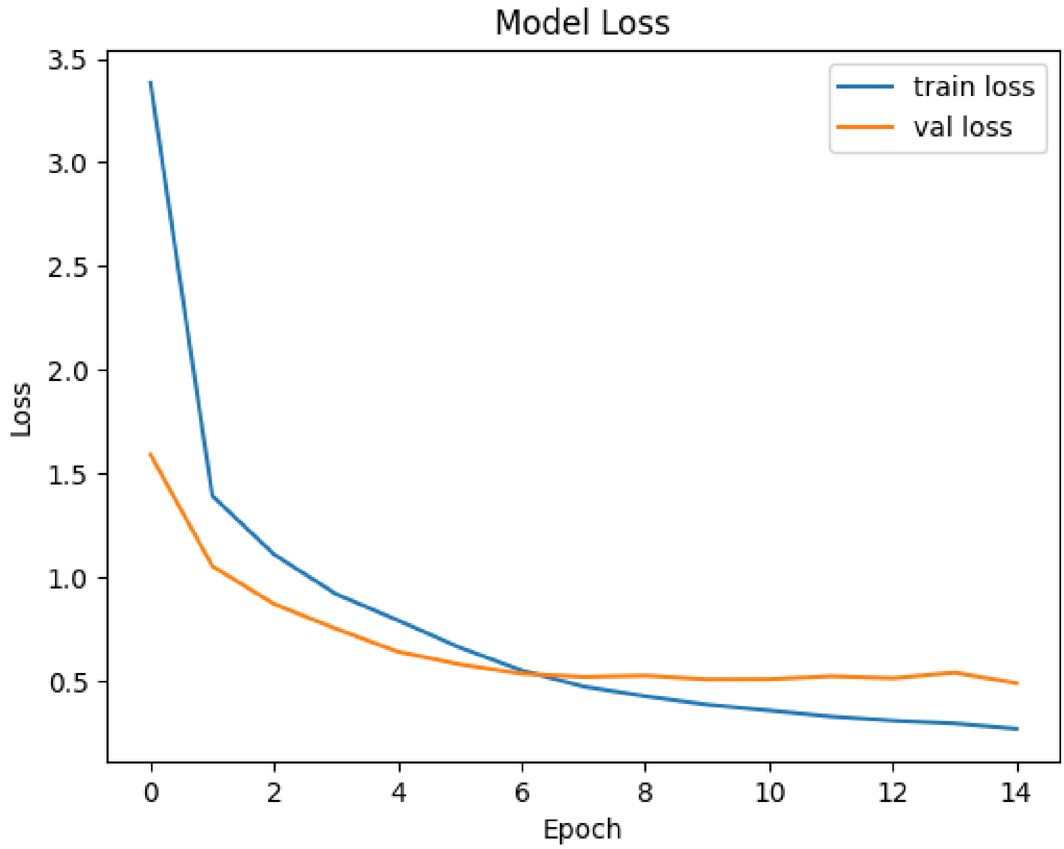
Appendix 4.5 CNN Code Architecture

```
def cnn():

    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(128, 128, 1),
                               kernel_regularizer=tf.keras.regularizers.l2(0.001)),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Dropout(0.05),

        tf.keras.layers.Conv2D(64, (3, 3), activation='relu',
                               kernel_regularizer=tf.keras.regularizers.l2(0.001)),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Dropout(0.05),

        tf.keras.layers.Conv2D(128, (3, 3), activation='relu',
                               kernel_regularizer=tf.keras.regularizers.l2(0.001)),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Flatten(),

        tf.keras.layers.Dense(128, activation='relu',
                              kernel_regularizer=tf.keras.regularizers.l2(0.001)),
        tf.keras.layers.Dropout(0.15),
        tf.keras.layers.Dense(8, activation='softmax')
    ])
```

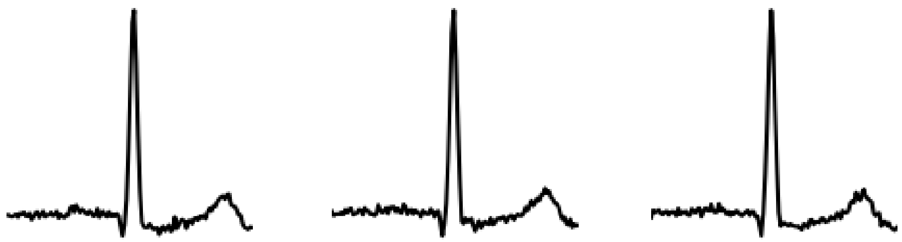
Appendix 4.6 CNN Model Accuracy



Appendix 4.7 CNN Model Loss



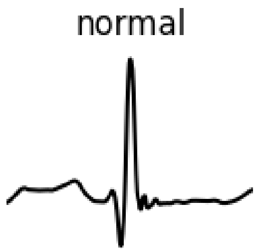
Appendix 4.8 AD8232 Acquired Data Images



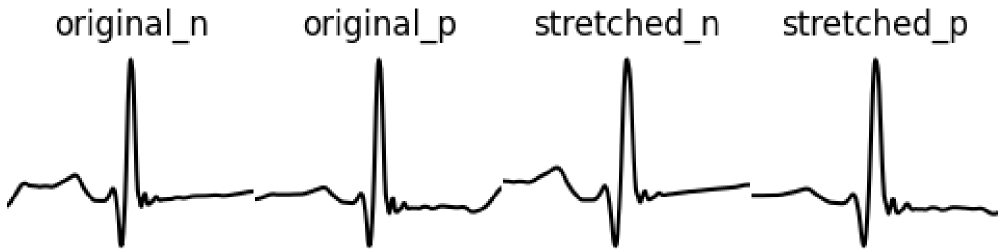
Appendix 4.9 Data Augmentation

Example of original image and 4 augmented counterparts:

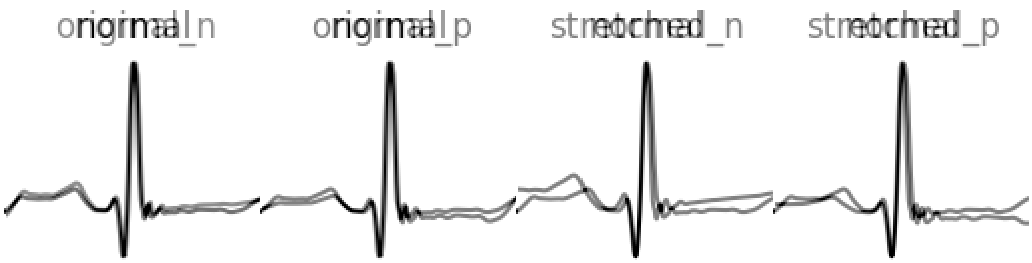
ORIGINAL:



AUGMENTATED:



STACKED VIEW OF NORMAL & EACH IMAGE:

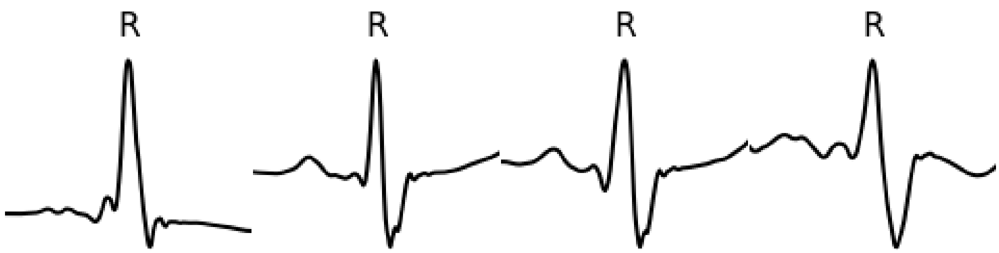


Appendix 4.10 Heartbeat Types Covered in this Project – Preprocessing Generated Images Examples of Each

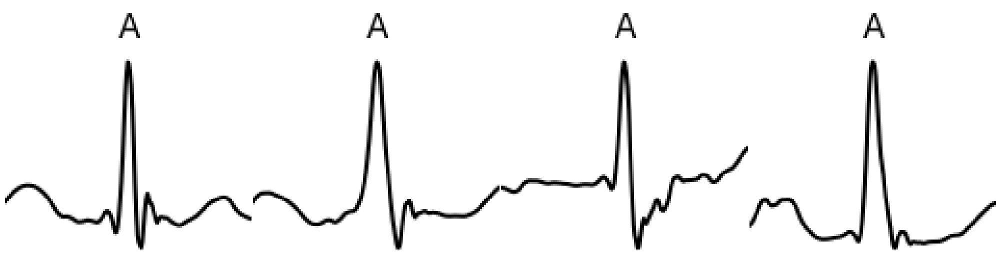
NORMAL:

LEFT BUNDLE BRANCH BEAT:

RIGHT BUNDLE BRANCH BEAT:



ATRIAL PREMATURE BEAT:



PREMATURE VENTRICULAR CONTRACTION:

VENTRICULAR ESCAPE BEAT:

VENTRICULAR FLUTTER WAVE:

PACED BEAT:

Appendix 5 Code Developed

Appendix 5.1 GitHub Repository Link

https://github.com/EthanLynam/arrythmia_detector_cnn

Appendix 5.2 generate_images.py lines 147-150

```
if __name__ == '__main__':
    with multiprocessing.Pool(processes=os.cpu_count()) as pool:
        pool.map(process_patient_record, records)
```

Appendix 5.3 generate_images.py lines 45-64

```
def process_patient_record(patient_num):

    print(f"Processing record: {patient_num}")

    record_data = wfdb.rdsamp(f'data/mit_bih_records/{patient_num}')
     annotation  =  wfdb.rdann(f'data/mit_bih_records/{patient_num}',
'atr')

    ecg_data = record_data[0].transpose()

    # Two lead types to choose from.
    # This will be MLII for all records but 102, 104,
    # who's data is clearer on second lead (both on pacemakers)
    if patient_num == 114 or 207:
        ecg_data = ecg_data[1]
    else:
        ecg_data = ecg_data[0]

    ecg_data = denoise_signal(ecg_data) # Denoise the data
     ecg_data  =  remove_baseline_wander(ecg_data)  #  Remove  baseline
wander
    rpeaks_indices = detect_rpeaks(ecg_data, SAMPLING_RATE) # identify
r peaks
```

Appendix 5.4 generate_images.py lines 68-111

```
 # This part of the code identifies noisy periods to be skipped later.
    noisy_periods = []  # List to store noisy periods
    SKIP_NOISY_PERIOD = False

     #  Loop  through  all  annotations  and  identify  the  noisy  periods
based on subtypes
    for i, annotation_sample in enumerate(annotation.sample):
        annotation_symbol = annotation.symbol[i]
        subtype = annotation.subtype[i]

        # Check for noisy periods based on subtype
```

```
        if (annotation_symbol == '~') and (subtype in [1, 2, 3]):  #
Start of noisy period

            # Only mark the start of the noisy period if we aren't
already in one
            if not SKIP_NOISY_PERIOD:

                SKIP_NOISY_PERIOD = True
                start_sample = annotation_sample

        elif (annotation_symbol == '~') and (subtype == 0):  # Clean
period (end of noisy period)

            # Only mark the end of the noisy period if we are currently
in one
            if SKIP_NOISY_PERIOD:

                SKIP_NOISY_PERIOD = False
                                 noisy_periods.append((start_sample,
annotation_sample))

        elif (annotation_symbol == '~') and (subtype == -1):
            continue


    for idx, rpeak_idx in enumerate(rpeaks_indices['rpeaks']):

        # Check if the current R-peak is within a noisy period
        IN_NOISY_PERIOD = False
        for start_sample, end_sample in noisy_periods:
            if start_sample <= rpeak_idx <= end_sample:
                IN_NOISY_PERIOD = True
                break

        # If we're in a noisy period, skip the heartbeat
        if IN_NOISY_PERIOD:
              print(f' Patient {patient_num}: {idx} SKIPPED - NOISY
DATA.')
            continue
```

Appendix 5.5 generate_images.py lines 114-144

```
        # Find the annotation closest to the R-peak
                                         closest_annotation_idx        =
numpy.argmin(numpy.abs(annotation.sample - rpeak_idx))
       closest_annotation = annotation.symbol[closest_annotation_idx]
         full_ann = ann_translate.get(closest_annotation, 'OTHER') #
use ann map to translate

        # Define the range for the current beat (centered around the
R-peak)
        start_idx = max(rpeak_idx - PRE_R_WINDOW, 0)
        end_idx = min(rpeak_idx + POST_R_WINDOW, len(ecg_data))
```

```
        beat = ecg_data[start_idx:end_idx] # Extract the heartbeat
segment

        # create the directory if it doesnt exist for future users
        os.makedirs(f'{IMAGES_PATH}/{full_ann}', exist_ok=True)

        # Create the plot
        # 3.31, 3.04 for 256 x 256 sized image
        fig, ax = plt.subplots(figsize=(1.66, 1.67), dpi=100)
        ax.plot(beat, color='black')
        ax.set_xlim(0, len(beat))
        ax.axis('off')

        # Save the figure in the created images folder for viewing
        fig.savefig(
            f'{IMAGES_PATH}/{full_ann}/{patient_num}_{idx}.png',
            dpi=100,
            bbox_inches=None,
            pad_inches=0
        )
          print(f' Patient {patient_num}: HEARTBEAT {idx} - IMAGE
CREATED.')

        # creates extra augmented images of any arrythmias
        beat_augment(beat, closest_annotation, idx, patient_num)
```

Appendix 5.6 augment_images.py lines 15-52

```
def beat_augment(beat_data, beat_label, idx, patient_num):
    """creates 4 slightly altered versions of beat_data"""

    if beat_label == 'N':
        return

    # take the original beat and stretch it by 1.25
    x_original = numpy.arange(len(beat_data))
        x_stretched = numpy.linspace(0, len(beat_data) - 1,
int(len(beat_data) * 1.25))
    interpolator = interp1d(x_original, beat_data, kind='linear')
    beat_stretched = interpolator(x_stretched)

     # recentre the stretched beats, as stretching moves them to the
right
    # on the x axis
    middle_idx = len(beat_stretched) // 2
    start_idx_edit = max(middle_idx - 128, 0)
    end_idx_edit = min(middle_idx + 128, len(beat_stretched))
    beat_stretched = beat_stretched[start_idx_edit:end_idx_edit]

    # polynomials for original beat
    x_original_poly = numpy.linspace(0, 1, len(beat_data))
    poly_positive = 0.3 * x_original_poly**2 + 0.7 * x_original_poly
+ 0.2
```

```
    poly_negative = -0.2 * x_original_poly**2 - 0.5 * x_original_poly
+ 0.8

    # polynomials for stretched beat
    x_stretched_poly = numpy.linspace(0, 1, len(beat_stretched))
     poly_positive_stretched = 0.4 * x_stretched_poly**2 + 0.9 *
x_stretched_poly + 0.3
     poly_negative_stretched = -0.3 * x_stretched_poly**2 - 0.6 *
x_stretched_poly + 0.7

     # multiply normal beat by positive and negative polynomials,
creating
    # two new images
    new_positive_beat = beat_data * poly_positive
    new_negative_beat = beat_data * poly_negative

    # multiply stretched beat by positive and negative polynomials,
creating
    # two other new images
           new_positive_beat_stretched = beat_stretched *
poly_positive_stretched
           new_negative_beat_stretched = beat_stretched *
poly_negative_stretched
```

Appendix 5.7 detect_rpeaks.py

```
from biosppy.signals import ecg

def detect_rpeaks(signal, sampling_rate):

    # identifies ecg features (including r peaks)
    general_data = ecg.ecg(
        signal=signal,
        sampling_rate=sampling_rate,
        show=False
    )

    return general_data['rpeaks']
```

Appendix 5.8 ecg_baseline_wander.py

```
"""
This module taes in ecg data and removes baseline wander,
that being the data 'wandering' and moving off course.
It utilises a butterworth filter and the linear
filtfilt function to return the data, now straightened
up with no loss of data shape.
"""

from scipy.signal import butter, filtfilt
```

```python
def remove_baseline_wander(ecg_data):
    """removes baseline wander from given signal"""

    samp_frequency = 360

    normal_cutoff = 0.5 / (0.5 * samp_frequency)

    b, a = butter(1, normal_cutoff, btype='high', analog=False)

    return filtfilt(b, a, ecg_data)
```

Appendix 5.9 ecg_denoise.py

```python
"""
This module takes in the ECG data and returns the signal in a
denoised form, by decomposing the signal into ten sub signals
and removing the low signal and the two highest signals.
The amiunt of signals being removed can be changed to
create greater effects of denoising. Finally it inverses the
wavelet transform which returns it to a signle signal.
"""

import pywt
import numpy

def denoise_signal(ecg_data):
    """denoises the given signal"""

    # decompose signal into wavelt coefficients
    coefficients = pywt.wavedec(ecg_data, 'db5', level=10)

    low_cutoff = 1
    high_cutoff = 8

    # zero out coeffocoents from 0 to low cutoff
    for num in range(0, low_cutoff):
        coefficients[num] = numpy.multiply(coefficients[num],[0.0])

    # zero out coefficients from high cutoff to end
    for num in range(high_cutoff, len(coefficients)):
        coefficients[num] = numpy.multiply(coefficients[num], 0)

    # inverse wavelet transform to reconstruct the signal, now denoised
    denoised_ecg = pywt.waverec(coefficients, 'db5')

    return denoised_ecg
```

Appendix 5.10 server.py: main()

```
async def main():

    # start websocket server on 0.0.0.0 and port 9000
    server = await websockets.serve(handle_connection, "0.0.0.0",
9000)
    print("Websocket server started on 0.0.0.0 & Port 9000...")

    # keep server running
    await server.wait_closed()
```

Appendix 5.11 server.py: process_data()

```
def process_data():

    global ecg_queue, counter, processing
    print("Processing data...")

    processing = True
    ecg_data = np.array(ecg_queue) # convert the queue to array

    ecg_data2 = denoise_signal(ecg_data) # denoise the signal
    ecg_data3 = remove_baseline_wander(ecg_data2) # remove baseline
wander
    r_peaks_indices = detect_rpeaks(ecg_data3) # detect R-peaks

    # generate images around detected peaks
    create_images(ecg_data, r_peaks_indices)

    for filename in os.listdir('data/created_images'):

        image_path = os.path.join('data/created_images', filename)
        classify_heartbeat(image_path)
        os.remove(image_path)

    # reset counter and clear queue
    counter = 0
    ecg_queue.clear()

    processing = False  # Reset flag after processing is done

    # force garbage collection to prevent error from matplotlib being
    # used in secondary thread
    gc.collect()

    print("Review results. Data queue cleared and receiving next
batch.")
```

Appendix 5.12 server.py: handle_connection()

```
async def handle_connection(websocket):

    global counter, ecg_queue, processing
    print("Client connected.")
    print("Receiving data...")

    try:
        async for message in websocket:

            ecg_value = int(message) # convert message to int (arduino
sending as string)
            ecg_queue.append(ecg_value)  # add data to the queue
            counter += 1 # increase counter

            # Although plenty of time is given to process data,
            # should the thread still be busy the data will just
            # continue to be recieved at the cost of some loss
            if counter >= 1625 and not processing:
                # start thread to process the data, as processing on
same
                # will create data loss as arduino continuously sends
data
                print("Queue full!")
                Thread(target=process_data).start()
                processing = True

    except websockets.exceptions.ConnectionClosedError:
        print() # This error follows straight after client connection,
        # However the connection remains, hence it is caught and
ignored.
```

Appendix 5.13 heartbeat_classification.py

```
# pylint: skip-file

import numpy as np
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image

model = load_model('../cnn/models/arrythmia_detection_cnn.keras')

class_labels = [
    "ATRIAL PREMATURE BEAT",
    "LEFT BUNDLE BRANCH BEAT",
    "NORMAL",
    "PACED BEAT",
    "PREMATURE VENTRICULAR CONTRACTION",
    "RIGHT BUNDLE BRANCH BEAT",
    "VENTRICULAR ESCAPE BEAT",
    "VENTRICULAR FLUTTER WAVE"
]
```

```
def classify_heartbeat(image_path):

    img = image.load_img(
        image_path,
        target_size=(128, 128),
        color_mode='grayscale'
    )

    image_array = image.img_to_array(img)
    image_array = np.expand_dims(image_array, axis=0)

    # Predict the class
    probability_predictions = model.predict(image_array)  # get probabilities for each class

    output = np.argmax(probability_predictions[0])  # index of highest probability = 0
    output_name = class_labels[output]  # use class_labels map for full name

    if output_name != 'NORMAL':
        print(
            "\n"
            "***************************************************\n"
            "*                                                 *\n"
            "*  ⚠️  WARNING: ABNORMAL HEARTBEAT DETECTED  ⚠️   *\n"
            "*                                                 *\n"
           f"*       TYPE: {output_name}        *\n"
            "*                                                 *\n"
            "***************************************************\n"
        )
    else:
        print("HEARTBEAT NORMAL - ALL CLEAR!")
```

Appendix 5.14 ecg_websocket_client.ino: void setup()

```
void setup() {
    Serial.begin(115200);
      while (!Serial) {
      ;  // wait for serial port to connect.
    }

    WiFi.begin(SSID, PASSWORD);
    Serial.print("Connecting to Wi-Fi...");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print(".");
        delay(300);
    }
    Serial.println();
    Serial.print("Connected to WiFi: ");
    Serial.println(SSID);
```

```
    // initialize WebSocket
    webSocket.begin(serverAddress, serverPort);

    // event handler
    webSocket.onEvent(webSocketEvent);
}
```

Appendix 5.15 ecg_websocket_client.ino: void loop()

```
void loop() {
    webSocket.loop();

    int ecgData = analogRead(A0);

    char dataToSend[20];
    sprintf(dataToSend, "%d", ecgData);
    webSocket.sendTXT(dataToSend);

    delay(3); //300hZ to mimick mit-bih 360Hz notch filter
}
```

Appendix 5.16 dataset_prep.py

```
'''
Takes the images created by preprocessing and creates links inside a
new folder 'data'
for CNN to use for training, validation and testing. Shuffles order
of files
before creating links to prevent training, val, test from containing
mostly images
from a single patient, and instead takes a balanced mix of patients
data.
It also only takes maximum 3000 images from each of NOR, APC, LBB etc.
to have a balanced distribution of data, due to NOR unfiltered having
70000
images vs others with less than 1000.
'''

import os
import random

# paths are created relative to location of cnn_notebook.ipynb
# as dataset_prep is executed there
SOURCE_DIR = "../preprocessing/data/created_images"
TEST_DIR = "data/test"
TRAINING_DIR = "data/training"
VALIDATION_DIR = "data/validation"
```

```
# ratios (70% training, 20% validation, 10% test)
TRAIN_RATIO = 0.7
VAL_RATIO = 0.2
TEST_RATIO = 0.1

def dataset_prep():
    if os.path.exists("data"):
        print("Data folder already exists. Skipping folder creation.")
        return

    # create output folders if they dont exist
    os.makedirs(TRAINING_DIR, exist_ok=True)
    os.makedirs(VALIDATION_DIR, exist_ok=True)
    os.makedirs(TEST_DIR, exist_ok=True)

    # iterate through folders in created_images
    for class_folder in os.listdir(SOURCE_DIR):

        # avpids OTHER as this is for viewing purposes,
        # we are not interested in training the CNN to detect OTHER,
        # which contains other notable heart activites
        if class_folder in ["OTHER"]:
            continue

        class_path = os.path.join(SOURCE_DIR, class_folder) # creates
path for current folder in created-images

        # all images in current folder
        images = os.listdir(class_path)
        random.shuffle(images)  # shuffle the images in memory to
prevent bunvhes of same patients data

        # downsamples any data that has more than 3000 images,
        # to accomodate the smaller sets of data (NOR = 70000 while
VEB = 500)
        if len(images) > 3000:
            images = random.sample(images, 3000)

        # find splits (70 and 20 percent, remainder 10)
        total_images = len(images)
        train_end = int(TRAIN_RATIO * total_images)
        val_end = train_end + int(VAL_RATIO * total_images)

        # Split images into train, val, and test
        train_images = images[:train_end]
        val_images = images[train_end:val_end]
        test_images = images[val_end:]

        # used to associate the segment types with their folder
directories
        folder_map = {
            TRAINING_DIR: train_images,
            VALIDATION_DIR: val_images,
            TEST_DIR: test_images
        }
```

```
        # loops through folder_map
        for folder_name, segment_images in folder_map.items():

            # loops through each file (image) in each segment (train, 
validate, test)
            for image in segment_images:

                # path to current image to be added, found in created-
images
                source_image_path = os.path.join(class_path, image)

                  # path to either (test, train, val)'s type folder 
(normal, paced etc.)
                     destination_folder = os.path.join(folder_name, 
class_folder)
                   os.makedirs(destination_folder, exist_ok=True) # 
creates folder if doesnt exist (for first run on machine)

                  # joins destination folder to a file name (source 
images name found in created-images)
                 destination_path = os.path.join(destination_folder, 
image)

                # links image inside created-image to new destination 
inside test train val,
                # which prevents copying the images and instead holds 
the same data in 2 places.
                try:
                    os.link(source_image_path, destination_path)
                except FileExistsError:
                    pass

    print("Created_images dataset successfully downsized, shuffled & 
split into training, validation, and test sets using file links.")
```

Appendix 5.17 Code Snippet 1

```
ecg_data = record_data[0].transpose()

    # Two lead types to choose from.
    # This will be MLII for all records but 102, 104,
    # who's data is clearer on second lead (both on pacemakers)
    if patient_num == 114 or 207:
        ecg_data = ecg_data[1]
    else:
        ecg_data = ecg_data[0]
```

Appendix 5.18 Code Snippet 2

```
# This part of the code identifies noisy periods to be skipped later.
    noisy_periods = []  # List to store noisy periods
    SKIP_NOISY_PERIOD = False

     # Loop through all annotations and identify the noisy periods
based on subtypes
    for i, annotation_sample in enumerate(annotation.sample):
        annotation_symbol = annotation.symbol[i]
        subtype = annotation.subtype[i]

        # Check for noisy periods based on subtype
        if (annotation_symbol == '~') and (subtype in [1, 2, 3]):  #
Start of noisy period

             # Only mark the start of the noisy period if we aren't
already in one
            if not SKIP_NOISY_PERIOD:

                SKIP_NOISY_PERIOD = True
                start_sample = annotation_sample

        elif (annotation_symbol == '~') and (subtype == 0):  # Clean
period (end of noisy period)

            # Only mark the end of the noisy period if we are currently
in one
            if SKIP_NOISY_PERIOD:

                SKIP_NOISY_PERIOD = False
                                 noisy_periods.append((start_sample,
annotation_sample))

        elif (annotation_symbol == '~') and (subtype == -1):
            continue
```

……………

```
# Check if the current R-peak is within a noisy period
        IN_NOISY_PERIOD = False
        for start_sample, end_sample in noisy_periods:
            if start_sample <= rpeak_idx <= end_sample:
                IN_NOISY_PERIOD = True
                break

        # If we're in a noisy period, skip the heartbeat
        if IN_NOISY_PERIOD:
              print(f' Patient {patient_num}: {idx} SKIPPED - NOISY
DATA.')
            continue
```

Appendix 5.19 Code Snippet 3

```
'''
Takes the images created by preprocessing and creates links inside a
new folder 'data'
for CNN to use for training, validation and testing. Shuffles order
of files
before creating links to prevent training, val, test from containing
mostly images
from a single patient, and instead takes a balanced mix of patients
data.
It also only takes maximum 3000 images from each of NOR, APC, LBB etc.
to have a balanced distribution of data, due to NOR unfiltered having
70000
images vs others with less than 1000.
'''

import os
import random

# paths are created relative to location of cnn_notebook.ipynb
# as dataset_prep is executed there
SOURCE_DIR = "../preprocessing/data/created_images"
TEST_DIR = "data/test"
TRAINING_DIR = "data/training"
VALIDATION_DIR = "data/validation"

# ratios (70% training, 20% validation, 10% test)
TRAIN_RATIO = 0.7
VAL_RATIO = 0.2
TEST_RATIO = 0.1

def dataset_prep():
    if os.path.exists("data"):
       print("Data folder already exists. Skipping folder creation.")
        return

    # create output folders if they dont exist
    os.makedirs(TRAINING_DIR, exist_ok=True)
    os.makedirs(VALIDATION_DIR, exist_ok=True)
    os.makedirs(TEST_DIR, exist_ok=True)

    # iterate through folders in created_images
    for class_folder in os.listdir(SOURCE_DIR):

        # avpids OTHER as this is for viewing purposes,
        # we are not interested in training the CNN to detect OTHER,
        # which contains other notable heart activites
        if class_folder in ["OTHER"]:
            continue

        class_path = os.path.join(SOURCE_DIR, class_folder) # creates
path for current folder in created-images

        # all images in current folder
        images = os.listdir(class_path)
         random.shuffle(images)  # shuffle the images in memory to
prevent bunvhes of same patients data
```

```
        # downsamples any data that has more than 3000 images,
        # to accomodate the smaller sets of data (NOR = 70000 while
VEB = 500)
        if len(images) > 3000:
            images = random.sample(images, 3000)

        # find splits (70 and 20 percent, remainder 10)
        total_images = len(images)
        train_end = int(TRAIN_RATIO * total_images)
        val_end = train_end + int(VAL_RATIO * total_images)

        # Split images into train, val, and test
        train_images = images[:train_end]
        val_images = images[train_end:val_end]
        test_images = images[val_end:]

        # used to associate the segment types with their folder
directories
        folder_map = {
            TRAINING_DIR: train_images,
            VALIDATION_DIR: val_images,
            TEST_DIR: test_images
        }

        # loops through folder_map
        for folder_name, segment_images in folder_map.items():

            # loops through each file (image) in each segment (train,
validate, test)
            for image in segment_images:

                # path to current image to be added, found in created-
images
                source_image_path = os.path.join(class_path, image)

                # path to either (test, train, val)'s type folder
(normal, paced etc.)
                destination_folder = os.path.join(folder_name,
class_folder)
                os.makedirs(destination_folder, exist_ok=True) #
creates folder if doesnt exist (for first run on machine)

                # joins destination folder to a file name (source
images name found in created-images)
                destination_path = os.path.join(destination_folder,
image)

                # links image inside created-image to new destination
inside test train val,
                # which prevents copying the images and instead holds
the same data in 2 places.
                try:
                    os.link(source_image_path, destination_path)
                except FileExistsError:
                    pass
```

```
    print("Created_images dataset successfully downsized, shuffled &
split into training, validation, and test sets using file links.")
```