STRIPE YOUNG SCIENTIST AND TECHNOLOGY EXHIBITION 2026

Developing a Compression Pipeline for deploying Neural Networks in Wearable Sleep Apnoea Detection.

Addison Carey

Stand Number: 5216

Project Report and Diary

Table of Contents

1 Abstract

Sleep apnoea is a common but often undiagnosed sleep disorder, where a person’s breathing repeatedly stops and starts during sleep. It is connected to serious health problems like cardiovascular disease, chronic fatigue, and reduced quality of life. Currently, diagnosis often requires overnight studies in specialist sleep laboratories. These are expensive, time-consuming, and inaccessible for many people. Wearable health devices have the potential to make continuous, at-home monitoring more practical and affordable, but major challenges remain. Deep learning models required to detect apnoea episodes are often too large and computationally demanding to run effectively on resource-limited devices such as microcontrollers or low-powered mobile processors.

Recently, Artificial Intelligence (AI) and Machine Learning, particularly in how deep learning models can be adapted for real-world applications has been central to my research. AI models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) can be trained to recognise patterns in breathing, heart rate, or oxygen saturation data that indicate apnoea events. However, these models typically require powerful hardware and cloud-based processing, which creates problems with latency, battery efficiency, and privacy. Neural network compression provides a solution to this problem. Compression techniques, such as pruning (removing redundant connections), quantisation (reducing precision of weights and activations), and knowledge distillation (training smaller models to replicate larger models) make it possible to deploy efficient yet accurate models directly onto wearable devices.

My project involves developing and compressing deep learning models to detect apnoea episodes from physiological signals, such as respiration or ECG data, and deploying them on wearable-class hardware. The aim is to design a pipeline where a baseline model is trained on medical datasets, then compressed using pruning, quantisation and distillation methods, before ported onto a microcontroller or smartphone. This allows real-time detection of apnoea episodes during sleep without the need for cloud connectivity, improving accessibility and data privacy. By running the models directly on-device, the system minimises energy consumption, reduces reliance on external infrastructure, and ensures that sensitive health data remains secure.

I started my project by training simple baseline models on public datasets. Initial experiments with compression methods have been executed to explore how much the models can be reduced in size without significantly harming accuracy. Currently, I am working to make these models compatible with edge hardware using frameworks like TensorFlow Lite Micro, with the goal of creating a wearable prototype that can run real-time detection. This work has the potential to make sleep apnoea detection more accessible worldwide while advancing research into efficient deep learning on constrained devices.

2 Introduction

In recent years, sleep-related breathing disorders have emerged as a major public health concern, with obstructive sleep apnoea being one of the most prevalent and underdiagnosed conditions worldwide. Sleep apnoea is characterised by repeated interruptions in breathing during sleep, leading to oxygen desaturation, fragmented sleep, and excessive daytime fatigue (Jordan, McSharry, & Malhotra, 2014). Long-term untreated sleep apnoea has been strongly associated with serious health complications including cardiovascular disease, hypertension, stroke, and impaired cognitive performance (Punjabi, 2008). Despite its high prevalence, diagnosis remains limited due to the complexity, cost and accessibility of conventional clinical assessment methods.

The current optimal standard for diagnosing sleep apnoea is overnight polysomnography conducted in specialist sleep laboratories. While this is clinically effective, polysomnography is resource-intensive, expensive, and inconvenient for patients which often results in long waiting times and reduced diagnostic coverage (Berry, et al., 2017). These limitations have motivated increasing interest in alternative and more accessible diagnostic approaches. In particular, wearable and portable health monitoring devices have gained significant attention as a means of enabling continuous, at-home sleep monitoring while reducing reliance on clinical infrastructure (Pantelopoulos & Bourbakis, 2010).

Recent advancements in Artificial Intelligence (AI) and Machine Learning (ML) have demonstrated strong potential for automated sleep apnoea detection using physiological signals such as electrocardiogram (ECG), respiration, and oxygen saturation data. Deep learning models including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have shown the ability to learn complex temporal and morphological patterns in ECG signals that correlate with apnoea events (Faust, Hagiwara, Hong, Lih, & Acharya, 2018). These approaches often achieve high diagnostic accuracy and outperform traditional signal-processing based methods. However, these models typically require substantial computational resources, large memory capacity, and high energy consumption which makes them unsuitable for direct deployment in resource-constrained devices.

This presents a significant challenge for real-world implementation. Many existing AI-based sleep apnoea detection systems rely on cloud-based processing which introduces issues related to latency, battery consumption, reliability, and data privacy. Transmitting sensitive physiological data to external servers raises concerns regarding security and patient confidentiality, particularly in continuous health monitoring applications (Li, Cheng, Wang, Morstatter, & Trevino, 2018). As a result, there is a growing need for AI models that can operate efficiently and securely directly on-device.

Neural Network compression has emerged as a promising solution to this challenge. Compression techniques such as pruning, quantisation, and knowledge distillation aim to reduce the size, computational complexity and energy requirements if deep learning models while preserving their predictive performance (Han, Mao, & Dally, 2016). By removing redundant parameters, reducing numerical precision or transferring knowledge from large models to smaller ones, compressed neural networks can be deployed on low-power hardware such as microcontrollers and mobile processors. These methods are increasingly central to the field of edge AI, where intelligence is embedded directly into devices rather than relying on cloud infrastructure.

The motivation for this project lies in developing a structured compression pipeline for neural networks applied to sleep apnoea detection, which a particular focus on wearable and edge-device deployment. The project begins by training a baseline deep learning model on publicly available ECG datasets, which establishes a reference level of diagnostic accuracy and computational cost. This model is then systematically compressed using pruning, quantisation and distillation techniques and evaluated in terms of accuracy, model size and efficiency. The ultimate aim is to enable real-time, on-device detection of apnoea events without reliance on cloud connectivity.

This project does not claim to replace clinical diagnostic tools or operate at a medical certification level. It instead seeks to investigate how far deep learning models for sleep apnoea detection can be reduced in size and complexity while remaining effective. Establishing this balance between accuracy and efficiency is essential for advancing practical health technologies. By focusing on compression-aware model design and deployment feasibility, this work contributes to ingoing research into efficient AI for healthcare applications.

Beyond technical objectives, this project reflects the broader challenge in modern AI research which is bridging the gap between high-performing models developed in controlled environments and systems that function reliably under real-world constraints. While deep neural networks continue to achieve remarkable performance across medical signal analysis tasks, their practical deployment often requires careful optimisation and compromise. By exploring compression strategies within the context of sleep apnoea detection, this project aims to demonstrate how intelligent systems can be adapted for real-world, low-power healthcare monitoring.

The proposed hypothesis is that through the application of structured neural network compression techniques, it is possible to significantly reduce the computational and memory requirements of deep learning models for sleep apnoea detection while maintaining clinically meaningful levels of accuracy, thereby enabling effective development of wearable-class hardware.

3 Explanations of Components and Terms

In this section of my report book, you will find explanations of each primary component of the Report Book. These Primary Components include Sleep Apnoea, Neural Networks, and Compression Techniques (Pruning, Quantisation, Distillation).

3.1 Sleep Apnoea

Sleep apnoea us a sleep-related breathing disorder characterised by repeated interruptions in normal breathing during sleep. These interruptions which are known as apnoeic events, occur when airflow stops (apnoea) or becomes significantly reduced (hypopnoea) for at least ten seconds. The most common form is Obstructive Sleep Apnoea (OSA), where the airway collapses or becomes blocked during sleep despite ongoing respiratory effort (Jordan, McSharry, & Malhotra, 2014) (Fig. 1).

Each apnoeic event reduces the level of oxygen in the blood and often triggers brief awakenings from sleep, preventing the body from reaching deeper, restorative sleep stages. Over time, this leads to excessive daytime sleepiness, impaired concentration and an increased risk of cardiovascular disease, hypertension and metabolic disorders (Punjabi, 2008). Since these awakenings are often unconscious, many individuals remain unaware of the condition, contributing to widespread underdiagnosis.

Physiological signals such as airflow, blood oxygen saturation, and electrocardiogram (ECG) recordings change measurably during apnoea events. In particular, ECG signals reflect variations in heart rate and autonomic nervous system activity caused by oxygen deprivation and arousal responses. These measurable changes make ECG data suitable for automated sleep apnoea detection using ML techniques (Faust, Hagiwara, Hong, Lih, & Acharya, 2018).

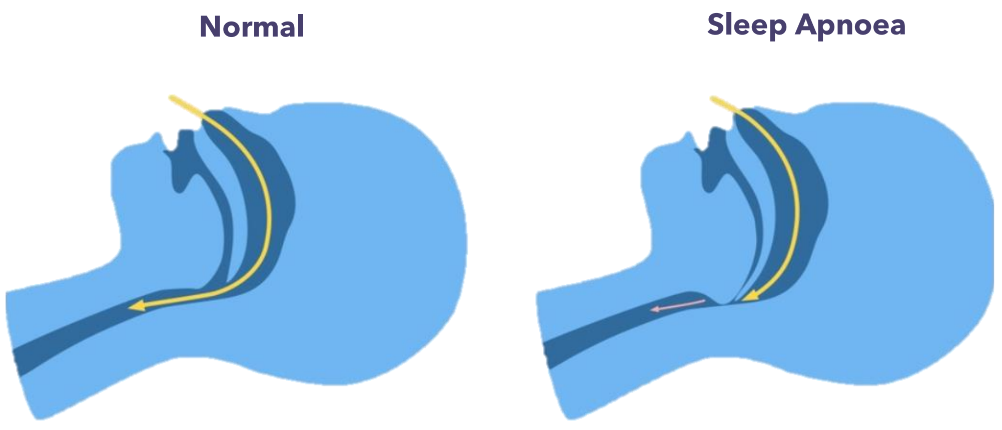


Figure 1: Normal VS Obstructed Airway comparison.

3.2 Neural Networks

Neural Networks (NNs) are a class of machine learning models inspired by the structure if the human brain. Theu consist of interconnected processing units called neurons, which are organised into layers: an input layer, one or more hidden layers, and an output layer (Fig. 2). Each neuron applies a weighted sum to its inputs, followed by a non-linear activation function, allowing the network to learn complex patterns in data (Goodfellow, Bengio, & Courville, 2016).

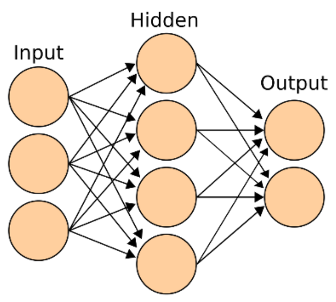


Figure 2: Illustration of the topology of a generic artificial NN.

In relation to sleep apnoea detection, NNs can be used to analyse physiological signals like ECG recordings. These signals contain subtle temporal and amplitude-based patterns that are difficult to model using traditional rude-based methods. NNs can automatically learn these patterns directly from data, enabling accurate classification of apnoea and normal breathing episodes (Faust, Hagiwara, Hong, Lih, & Acharya, 2018).

During training, NNs adjust their weights using optimisation algorithms such as gradient descent, minimising a loss function that measures the difference between predicted and true labels. While deeper and larger networks generally achieve higher accuracy, they also require more memory, computational power, and energy. This creates challenges when deploying neural networks on wearable or edge devices with limited resources.

3.3 Compression Techniques (Pruning, Quantisation, Distillation)

Neural network compression refers to a collection of techniques used to reduce the size, computational complexity and energy consumption of deep learning models without significantly degrading their performance. Compression is particularly important for deploying models on wearable and embedded devices, where memory and processing power are severely limited (Han, Mao, & Dally, 2016).

1. Pruning:

Pruning removed redundant or unimportant connections (weights) from a neural network. Many trained networks contain parameters that contribute very little to the final prediction. By identifying and removing these parameters, the model becomes smaller and faster while retaining most of its accuracy. Pruning can be unstructured, where individual weights are removed, or structured where entire neurons or layers are eliminated (Han, Mao, & Dally, 2016).

2. Quantisation:

Quantisation reduces the numerical precision of a model's weights and activations. Instead of using 32-bit floating-point values, quantised models may use 16-bit, 8-bit or even integer-only representations. This significantly reduces memory usage and allows models to run more efficiently on low-power hardware often with minimal impact on accuracy (Jacob, et al., 2018).

3. Knowledge Distillation:

Knowledge distillation involves training a smaller "student" model to mimic the behaviour of a larger, more accurate "teacher" model. Rather than learning directly from hard labels alone, the student model learns from the teacher's output probabilities, capturing richer information about the decision boundaries. This allows compact models to achieve performance close to much larger networks (Hinton, Vinyals, & Dean, 2015).

Together, these compression techniques form a pipeline that enables deep learning models to be implemented directly on-device, improving privacy, reducing latency, and lowering energy consumption in wearable health monitoring systems.

4 Literature Review

This literature review explores existing research into neural networks, model compression, and sleep apnoea detection. It is divided into four thematic sections: theoretical foundations of NNs, NN compression approaches, current sleep apnoea diagnosis approaches, and the identified research gap. These sections provide the academic context for this project and justify the need to efficient, deployable deep learning models in wearable sleep apnoea detection.

4.1 Theoretical Foundations of NNs

- Definition: NNs are interconnected processing systems capable of learning input-output relationships through adaptive weight updates, enabling them to model complex non-linear functions. (Haykin, 1999)
- Importance: They are particularly effective when large datasets are available, as their performance improves with exposure to diverse and representative training examples. They have demonstrated strong predictive performance across tasks such as classification, regression, and pattern recognition. (Goodfellow, Bengio, & Courville, 2016)
- Relevance: They operate on numerical representations stored in vectors, making them suitable for analysing real-works data which can include images, audio signals, text and time-series information. Physiological signals like ECG recordings are represented naturally as time-series data, allowing neural networks to learn temporal and amplitude-based patterns relevant to medical diagnosis (Faust, Hagiwara, Hong, Lih, & Acharya, 2018).

4.2 NN Compression Approaches

- Pruning: Pruning removes redundant of low-importance weights or neurons from a trained network. Studies have shown that many NNs are over-parameterised, and substantial portions of the network can be removed with minimal loss in accuracy (Han, Mao, & Dally, 2016).
- Quantisation: Quantisation reduces the numerical precision of model weights and activations, often from 32-bit floating-point to 8-bit or integer representations. This dramatically reduces memory usage and enables quicker inference on low-power hardware (Jacob, et al., 2018).
- Distillation: Knowledge Distillation involves training a smaller model to replicate the behaviour of a larger model. The smaller model learns from the soft output probabilities of the larger model, allowing compact models to achieve accuracy comparable to much larger networks (Hinton, Vinyals, & Dean, 2015).
- Observation: Compression methods often involve trade-offs between model size, accuracy and implementation complexity. Combining multiple compression techniques into a structured

pipeline has been shown to produce more efficient models suitable for edge deployment (Cheng, Wang, Zhou, & Zhang, 2018)

4.3 Current Sleep Apnoea Diagnosis Approaches

- The standard for diagnosing sleep apnoea is polysomnography (PSG), which is an overnight clinical test that records multiple physiological signals including EEG, ECG, airflow, oxygen saturation and respiratory effort. While PSG is highly accurate, it is expensive, time-consuming and requires specialised clinical facilities which limits accessibility for many patients (Punjabi, 2008).
- Traditional Approaches: Manual scoring of PSG recordings by trained clinicians and rule-based analysis of airflow and oxygen desaturation events.
- ML-based Approaches: Classical ML models like support vector machine and decision trees have been applied to extracted ECG and respiration features (Acharya, Chua, Faust, Lim, & Lim, 2011). Recently, deep learning approaches have demonstrated incredible performance by learning features directly from raw or minimally processed physiological signals (Faust, Hagiwara, Hong, Lih, & Acharya, 2018).
- Limitations of current AI-based systems: Many current proposed models rely on cloud-based processing due to high computational requirements. This introduces issues with power consumption, data privacy and latency. Very few studies address deployment on wearable or microcontroller-class hardware.

4.4 Identified Research Gap

From the reviewed literature, there are many key gaps that are identified:

1. High-performing deep learning models for sleep apnoea detection often require substantial computational resources.
2. Many studies prioritise classification accuracy without the consideration of deployability on wearable or edge devices.
3. Existing compression research is frequently evaluated on general benchmarks rather than medical time-series data.
4. End-to-end compression pipelines tailored specifically for wearable sleep apnoea detection remain underexplored.

This project aims to address these gaps by developing a structured NN compression pipeline specifically designed for deployment in wearable sleep apnoea monitoring systems.

4.5 Problem Statement and Aim

Despite advances in deep learning for sleep apnoea detection, most existing models remain impractical for real-world wearable development due to their size, power requirements and reliance on cloud infrastructure. While NN compression techniques exist, their combined application to biomedical time-series data has not been sufficiently explored.

Problem Statement:

Can a structured NN compression pipeline that incorporates pruning, quantisation and knowledge distillation produce an efficient and accurate sleep apnoea detection model suitable for deployment on wearable or edge devices without reliance on cloud-based processing?

The aim of this project is to design, implement and evaluate a NN compression pipeline for sleep apnoea detection using available ECG data. The project seeks to balance classification accuracy with computational efficiency to enable real-time, on-device inference in wearable systems.

4.6 Engineering and Design Goals

1. Baseline Model Development: Train an accurate NN model for sleep apnoea detection using publicly available ECG data.
2. Compression Pipeline Design: Apply pruning, quantisation and knowledge distillation to reduce model size and computational cost.
3. Performance Evaluation: Compare compressed and uncompressed models in terms of accuracy, memory usage, and inference efficiency.
4. Edge Compatibility: Adapt the model for deployment on wearable or resource-constrained hardware platforms.
5. Practical Relevance: Demonstrate the feasibility of real-time, privacy-preserving sleep apnoea detection without cloud dependence.

5 Method

The methodology for this project was designed to systematically develop, test, and refine an algorithm pipeline for compressing neural networks for wearable sleep apnoea detection by progressing through a series of iterative models. Each iteration incorporated improvements based on observed performance, culminating in a final working model and prototype with accurate predictions.

This project was created using the principles of User-Centred Design (UCD). This is a design approach that focuses on creating products, services, and experiences that are centred around the needs and desires of the people who will use them. To achieve this, UCD relies on a set of principles that guide the design process. These principles include:

- Empathy: Understanding the needs, goals, and motivations of the users and putting oneself in their shoes.
- Collaboration: Involving users in the design process and collaborating with them to co-create solutions.
- Iteration: Prototyping and testing ideas quickly and repeatedly to refine and improve the design.
- Inclusivity: Designing for the diverse needs and abilities of all potential users.
- Accessibility: Creating products and services that are accessible and usable by people with disabilities.
- User testing: Gathering feedback from users and using it to inform and improve the design.

UCD was applied by following the Agile methodology cycle (Figure 3), typical of computational engineering projects. Agile methodology is a project management approach that enables flexibility and empowers practitioners to engage in rapid iteration. It is based on the Agile Manifesto (Fowler, 2001), a set of values and principles for software development that prioritize the needs of the customer, the ability to respond to change, and the importance of delivering working software regularly. Agile methodologies such as Scrum and Lean, are designed to help teams and individuals deliver high-quality products in a fast-paced and dynamic environment by breaking down complex projects into smaller, more manageable chunks and continually reassessing and adjusting the project plan as needed. Agile teams are typically self-organising and cross-functional and rely on regular communication and collaboration to achieve their goals.

1. Initial prototypes were created to establish a functional baseline.
2. Successive refinements were introduced, tested, and evaluated.
3. A final version was compared against a benchmark brute-force solver to measure progress toward the engineering and design goals.

All code was written in Python and implemented in a high-level programming environment, while mathematical reasoning and worked examples were used to validate algorithmic choices.

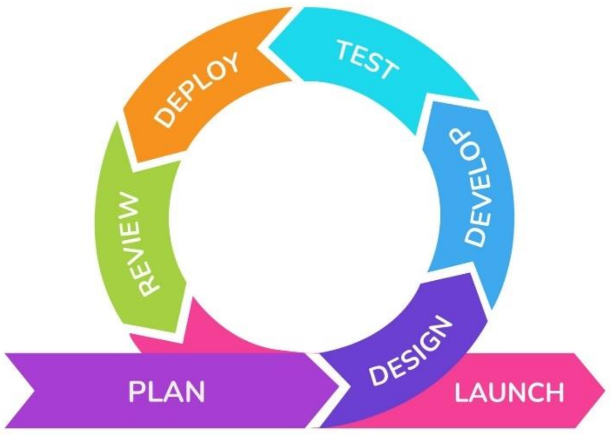


Figure 3: Agile Methodology Cycle

5.1 Procedure

The procedure involved four main phases: Planning and Designing, Developing, Testing, and Evaluation. Each iteration progressively advanced the capability of the model.

5.1.1 Iteration 1

Plan and Design Rationale

The purpose of Iteration 1 was to establish a baseline NN model for sleep apnoea detection using ECG data. This baseline model provided a reliable reference point against which future compression techniques and deployment-oriented optimisations could be evaluated in subsequent iterations (Fig. 4).

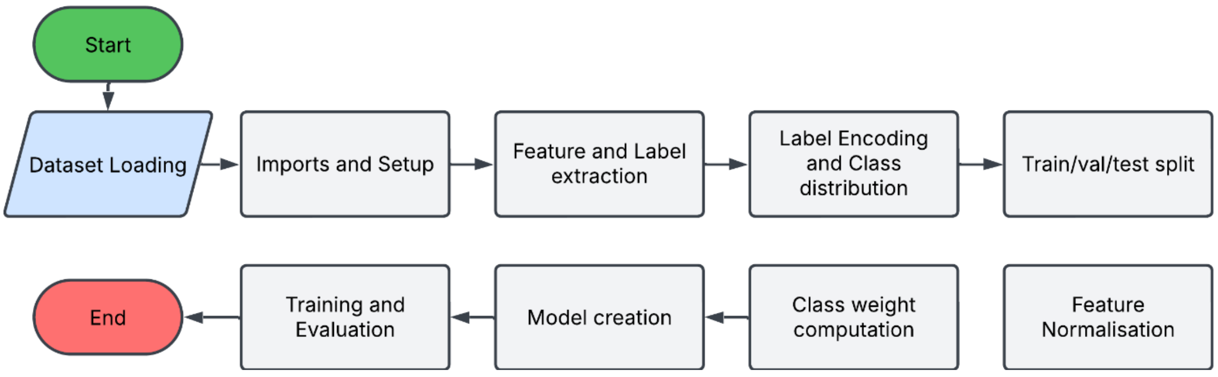


Figure 4: Flowchart of iteration 1 code design.

Development

The Iteration 1 program was written as a structured Python script implementing standard ML and deep learning libraries. Below is a breakdown of its main components:

1. Imports and Library Setup:

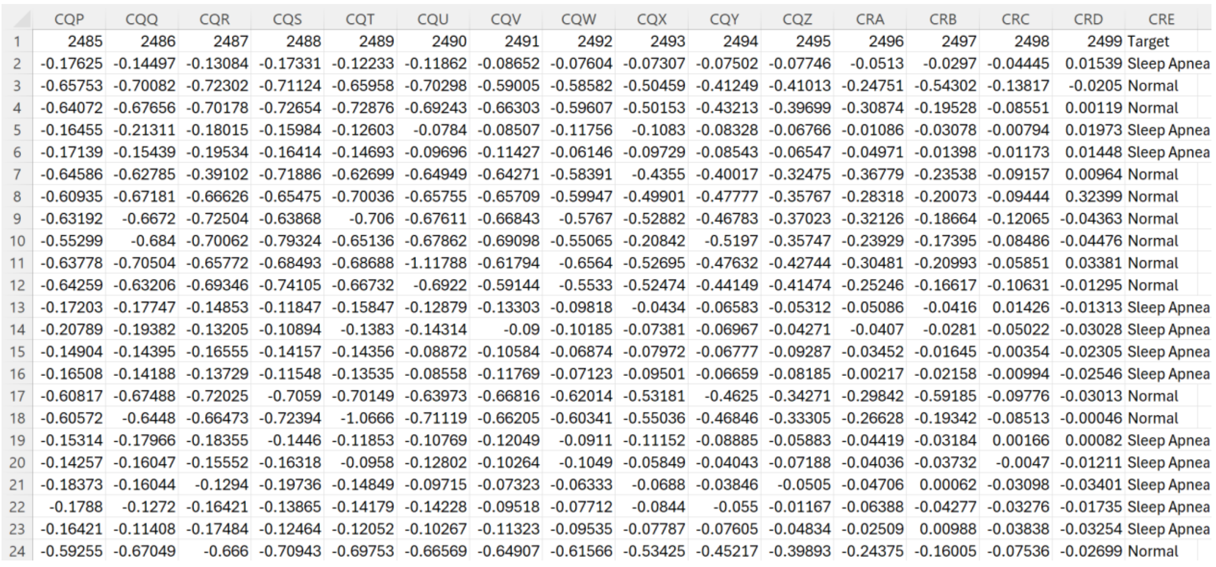
```python
#imports and setup AC
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils.class_weight import compute_class_weight
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
```

In these lines of code, the required libraries are imported for numerical computation, dataset handline, model training and evaluation. *NumPy* is used for numerical operations and label processing. *Pandas* provides tools for loading and manipulating the CSV dataset. *Scikit-learn* is used for dataset-splitting, feature normalisation, and handling class imbalance. Lastly, *TensorFlow Keras* is used to construct, train and save the NN model.

2. Dataset Loading:

```python
# path to data and read csv AC
dp = "ecg_sleep_apnea_dataset.csv"
df = pd.read_csv(dp)

print("Dataset shape:", df.shape)
print(df.head())
```

The dataset is loaded form a CSV file containing ECG feature vectors and class labels. The Kaggle ECG Sleep Apnoea dataset was chosen and consists of pre-segmented ECG signals represented as fixed-length numerical vectors which allowed for straightforward NN training. The print statements were used to check the dataset dimensions, correct placement of the target column and the absence of obvious formatting errors. Each row in the dataset represents one ECG segment, with numerical signal values followed by a target class label.



	CQP	CQQ	CQR	CQS	CQT	CQU	CQV	CQW	CQX	CQY	CQZ	CRA	CRB	CRC	CRD	CRE
1	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495	2496	2497	2498	2499	Target
2	-0.17625	-0.14497	-0.13084	-0.17331	-0.12233	-0.11862	-0.08652	-0.07604	-0.07307	-0.07502	-0.07746	-0.0513	-0.0297	-0.04445	0.01539	Sleep Apnea
3	-0.65753	-0.70082	-0.72302	-0.71124	-0.65958	-0.70298	-0.59005	-0.58582	-0.50459	-0.41249	-0.41013	-0.24751	-0.54302	-0.13817	-0.0205	Normal
4	-0.64072	-0.67656	-0.70178	-0.72654	-0.72876	-0.69243	-0.66303	-0.59607	-0.50153	-0.43213	-0.39699	-0.30874	-0.19528	-0.08551	0.00119	Normal
5	-0.16455	-0.21311	-0.18015	-0.15984	-0.12603	-0.0784	-0.08507	-0.11756	-0.1083	-0.08328	-0.06766	-0.01086	-0.03078	-0.00794	0.01973	Sleep Apnea
6	-0.17139	-0.15439	-0.19534	-0.16414	-0.14693	-0.09696	-0.11427	-0.06146	-0.09729	-0.08543	-0.06547	-0.04971	-0.01398	-0.01173	0.01448	Sleep Apnea
7	-0.64586	-0.62785	-0.39102	-0.71886	-0.62699	-0.64949	-0.64271	-0.58391	-0.4355	-0.40017	-0.32475	-0.36779	-0.23538	-0.09157	0.00964	Normal
8	-0.60935	-0.67181	-0.66626	-0.65475	-0.70036	-0.65755	-0.65709	-0.59947	-0.49901	-0.47777	-0.35767	-0.28318	-0.20073	-0.09444	0.32399	Normal
9	-0.63192	-0.6672	-0.72504	-0.63868	-0.706	-0.67611	-0.66843	-0.5767	-0.52882	-0.46783	-0.37023	-0.32126	-0.18664	-0.12065	-0.04363	Normal
10	-0.55299	-0.684	-0.70062	-0.79324	-0.65136	-0.67862	-0.69098	-0.55065	-0.20842	-0.5197	-0.35747	-0.23929	-0.17395	-0.08486	-0.04476	Normal
11	-0.63778	-0.70504	-0.65772	-0.68493	-0.68688	-1.11788	-0.61794	-0.6564	-0.52695	-0.47632	-0.42744	-0.30481	-0.20993	-0.05851	0.03381	Normal
12	-0.64259	-0.63206	-0.69346	-0.74105	-0.66732	-0.6922	-0.59144	-0.5533	-0.52474	-0.44149	-0.41474	-0.25246	-0.16617	-0.10631	-0.01295	Normal
13	-0.17203	-0.17747	-0.14853	-0.11847	-0.15847	-0.12879	-0.13303	-0.09818	-0.0434	-0.06583	-0.05312	-0.05086	-0.0416	0.01426	-0.01313	Sleep Apnea
14	-0.20789	-0.19382	-0.13205	-0.10894	-0.1383	-0.14314	-0.09	-0.10185	-0.07381	-0.06967	-0.04271	-0.0407	-0.0281	-0.05022	-0.03028	Sleep Apnea
15	-0.14904	-0.14395	-0.16555	-0.14157	-0.14356	-0.08872	-0.10584	-0.06874	-0.07972	-0.06777	-0.09287	-0.03452	-0.01645	-0.00354	-0.02305	Sleep Apnea
16	-0.16508	-0.14188	-0.13729	-0.11548	-0.13535	-0.08558	-0.11769	-0.07123	-0.09501	-0.06659	-0.08185	-0.00217	-0.02158	-0.00994	-0.02546	Sleep Apnea
17	-0.60817	-0.67488	-0.72025	-0.7059	-0.70149	-0.63973	-0.66816	-0.62014	-0.53181	-0.4625	-0.34271	-0.29842	-0.59185	-0.09776	-0.03013	Normal
18	-0.60572	-0.6448	-0.66473	-0.72394	-1.0666	-0.71119	-0.66205	-0.60341	-0.55036	-0.46846	-0.33305	-0.26628	-0.19342	-0.08513	-0.00046	Normal
19	-0.15314	-0.17966	-0.18355	-0.1446	-0.11853	-0.10769	-0.12049	-0.0911	-0.11152	-0.08885	-0.05883	-0.04419	-0.03184	0.00166	0.00082	Sleep Apnea
20	-0.14257	-0.16047	-0.15552	-0.16318	-0.0958	-0.12802	-0.10264	-0.1049	-0.05849	-0.04043	-0.07188	-0.04036	-0.03732	-0.0047	-0.01211	Sleep Apnea
21	-0.18373	-0.16044	-0.1294	-0.19736	-0.14849	-0.09715	-0.07323	-0.06333	-0.0688	-0.03846	-0.0505	-0.04706	0.00062	-0.03098	-0.03401	Sleep Apnea
22	-0.1788	-0.1272	-0.16421	-0.13865	-0.14179	-0.14228	-0.09518	-0.07712	-0.0844	-0.055	-0.01167	-0.06388	-0.04277	-0.03276	-0.01735	Sleep Apnea
23	-0.16421	-0.11408	-0.17484	-0.12464	-0.12052	-0.10267	-0.11323	-0.09535	-0.07787	-0.07605	-0.04834	-0.02509	0.00988	-0.03838	-0.03254	Sleep Apnea
24	-0.59255	-0.67049	-0.666	-0.70943	-0.69753	-0.66569	-0.64907	-0.61566	-0.53425	-0.45217	-0.39893	-0.24375	-0.16005	-0.07536	-0.02699	Normal

3. Feature and Label Extraction:

```python
# split features and labels AC
X = df.iloc[:, :-1].values
y_raw = df.iloc[:, -1].values  # string labels
```

These lines separate the dataset into:

- *X*: the ECG feature matrix (all columns except the last one)
- *y_raw*: the target labels (the final column)

This separation was necessary to allow the NN to learn a mapping from ECG features to diagnostic outcomes.

4. Label Encoding and Class Distribution

```python
# map string labels to integers AC
label_map = {
    "Normal": 0,
    "Sleep Apnea": 1
}

y = np.array([label_map[label] for label in y_raw])

print("Label distribution:", np.bincount(y))
```

NNs operate on numerical values, so categorical labels were converted into integers. Normal ECG segments are mapped to 1 and sleep apnoea segments are mapped to 1. This binary encoding aligns with the use of a sigmoid output neuron and binary cross-entropy loss. The print line calculates the number of samples belonging to each class. Evaluating the class balance at this stage is vital in medical classification problems, where biased datasets can lead to misleading accuracy metrics.

5. Train-Validation-Test Split:

```python
# train val test split AC
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y,
    test_size=0.30,
    random_state=42,
    stratify=y
)
```

The dataset is first split into training (70%) and temporary (30%) subsets using stratified sampling to preserve class proportions.

```python
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp,
    test_size=0.50,
    random_state=42,
    stratify=y_temp
)

print("Train:", X_train.shape)
print("Val:", X_val.shape)
print("Test:", X_test.shape)
```

The temporary subset is further divided evenly into validation and test sets (both 15%). This three-way split allows training on unseen data, hyperparameter monitoring via validation and an unbiased final evaluation using the test dataset.

6. Feature Normalisation:

```python
# data normalisation AC
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_val   = scaler.transform(X_val)
X_test  = scaler.transform(X_test)
```

Standardisation scales each feature to zero mean and unit variance. This improves convergence speed and training stability by preventing features with large magnitudes from dominating the optimisation process. The scalar is fit only on the training data to prevent information leakage

The lattice points were plotted in blue, while the shortest vector was displayed in red. This step confirmed visually that the algorithm was correctly identifying short lattice vectors.

7. Class Weight Computation:

```
# class weights to handle imbalance AC
classes = np.unique(y_train)

class_weights = compute_class_weight(
    class_weight="balanced",
    classes=classes,
    y=y_train
)

class_weights = dict(zip(classes, class_weights))
print("Class weights:", class_weights)
```

These lines compute the class weights inversely proportional to class frequency. This ensures that misclassification of underrepresented class incurs a higher penalty during training. Although the dataset is relatively balanced, this step improves robustness and aligns with best practice in biomedical ML.

8. Baseline Model Creation:

```
# baseline model creation AC
model = Sequential([
    Dense(128, activation="relu", input_shape=(X_train.shape[1],)),
    Dropout(0.3),

    Dense(64, activation="relu"),
    Dropout(0.3),

    Dense(1, activation="sigmoid")
])
```

The NN architecture consists of two fully connected hidden layers using ReLU activation, dropout layers to reduce overfitting, and a single sigmoid output neuron for binary classification. This design balances expressive power with simplicity, making it suitable as a baseline model.

```
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

model.summary()
```

The Adam optimiser is used due to its adaptive learning rate and stability across a wide range of problems. Binary cross-entropy is used as the loss function because the task is binary classification.

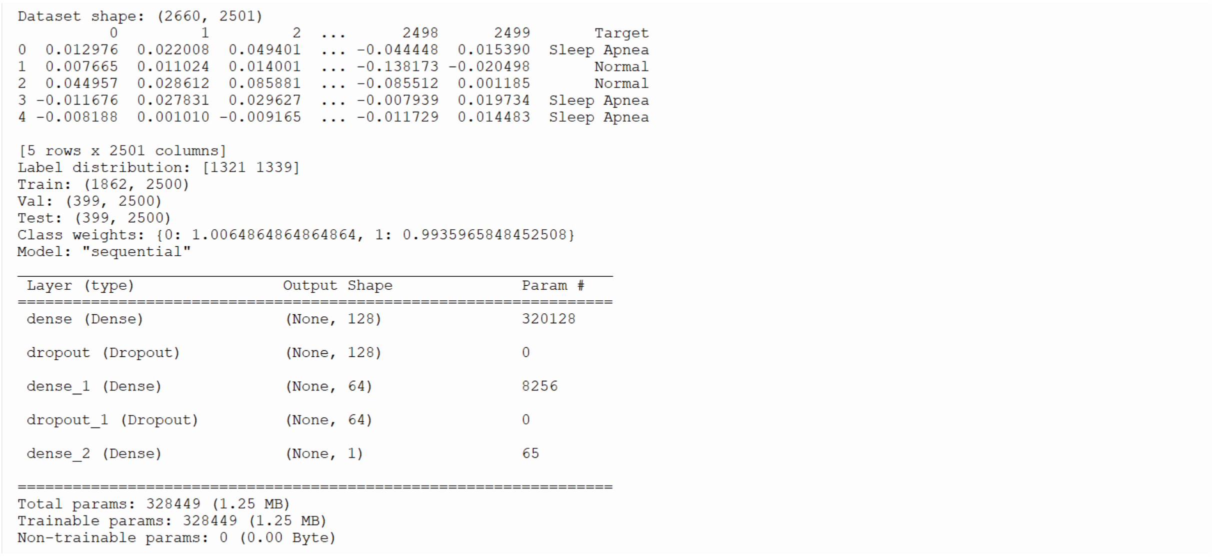
9. Model Training and Evaluation:

```
# baseline model training AC
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=25,
    batch_size=32,
    class_weight=class_weights
)

# model evaluation AC
loss, acc = model.evaluate(X_test, y_test)
print("Test accuracy:", acc)

# saved model AC
model.save("baseline_sleep_apnea_model.keras")
print("Baseline model saved.")
```

The model is trained over 25 epochs. Validation performance is monitored during training to identify potential overfitting. Class weights are applied to maintain balanced learning across diagnostic classes. The trained model is evaluated on the test set to provide an unbiased estimate of real-world performance. Lastly, the model is saved to allow reuse in later iterations.

Testing

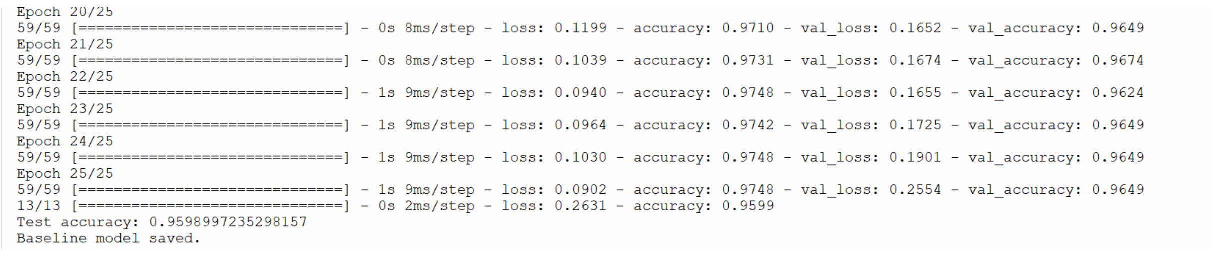
The baseline model was evaluated using a stratified train-validation-test split to preserve the distribution of normal and sleep apnoea samples (1321 normal, 1339 apnoea). The dataset consisted of 2660 ECG segments, each represented by 2500 features. Input features were standardised using z-score normalisation fitted on the training set (Fig 5).



```
Dataset shape: (2660, 2501)
          0         1         2  ...      2498      2499       Target
0  0.012976  0.022008  0.049401  ... -0.044448  0.015390  Sleep Apnea
1  0.007665  0.011024  0.014001  ... -0.138173 -0.020498       Normal
2  0.044957  0.028612  0.085881  ... -0.085512  0.001185       Normal
3 -0.011676  0.027831  0.029627  ... -0.007939  0.019734  Sleep Apnea
4 -0.008188  0.001010 -0.009165  ... -0.011729  0.014483  Sleep Apnea

[5 rows x 2501 columns]
Label distribution: [1321 1339]
Train: (1862, 2500)
Val: (399, 2500)
Test: (399, 2500)
Class weights: {0: 1.0064864864864864, 1: 0.9935965848452508}
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               320128

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 64)                8256

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 328449 (1.25 MB)
Trainable params: 328449 (1.25 MB)
Non-trainable params: 0 (0.00 Byte)
```

Figure 5: Output of Iteration 1 Model Structure and Parameters.

Training was performed over 25 epochs using class-weighted binary cross-entropy loss to mitigate mild class imbalance. Validation accuracy and loss were monitored at each epoch to assess convergence and generalisation behaviour. Model performance was then evaluated on a held-out test set (Fig 6).



```
Epoch 20/25
59/59 [==============================] - 0s 8ms/step - loss: 0.1199 - accuracy: 0.9710 - val_loss: 0.1652 - val_accuracy: 0.9649
Epoch 21/25
59/59 [==============================] - 0s 8ms/step - loss: 0.1039 - accuracy: 0.9731 - val_loss: 0.1674 - val_accuracy: 0.9674
Epoch 22/25
59/59 [==============================] - 1s 9ms/step - loss: 0.0940 - accuracy: 0.9748 - val_loss: 0.1655 - val_accuracy: 0.9624
Epoch 23/25
59/59 [==============================] - 1s 9ms/step - loss: 0.0964 - accuracy: 0.9742 - val_loss: 0.1725 - val_accuracy: 0.9649
Epoch 24/25
59/59 [==============================] - 1s 9ms/step - loss: 0.1030 - accuracy: 0.9748 - val_loss: 0.1901 - val_accuracy: 0.9649
Epoch 25/25
59/59 [==============================] - 1s 9ms/step - loss: 0.0902 - accuracy: 0.9748 - val_loss: 0.2554 - val_accuracy: 0.9649
13/13 [==============================] - 0s 2ms/step - loss: 0.2631 - accuracy: 0.9599
Test accuracy: 0.9598997235298157
Baseline model saved.
```

Figure 6: Output of Iteration 1 Code with Test Accuracy.

Evaluation

The fully connected neural network demonstrated rapid convergence, achieving validation accuracy of approximately 96.5% within the first few epochs. Training accuracy continued to improve gradually, reaching approximately 97.5% by the final epoch. Final evaluation on the test set yielded an accuracy of 95.99%, indicating strong generalisation performance.

The results show that even a simple dense architecture is capable of learning discriminative patterns from ECG-derived features. However, fluctuations in validation loss towards later epochs suggest the onset of overfitting, despite the use of dropout regularisation. This iteration therefore provided a strong performance baseline while highlighting architectural limitations.

Although effective, the baseline model treated each ECG segment as a flat feature vector and does not explicitly capture temporal structure inherent in physiological signals. The dense architecture also results in a relatively large parameter count (around 328,000 parameters) which limits efficiency and suitability for deployment on constrained devices. These limitations motivated the transition to convolutional architectures and model compression techniques in subsequent iterations.

5.1.2 Iteration 2

Plan and Design Rationale

Iteration 2 aimed to address the limitations identified in the baseline dense model by introducing a convolutional NN (CNN) architecture better suited to time-series ECG data. Unlike fully connected layers, 1D convolutions can learn local temporal patterns such as waveform morphology and rhythmic changes which are clinically relevant for sleep apnoea detection (Fig. 7,8).

Subsequently, magnitude-based weight pruning was introduced to reduce model complexity and move closer to a deployable solution. This aligned with the project's broader objective of developing an accurate yet resource-efficient model suitable for embedded or edge deployment.

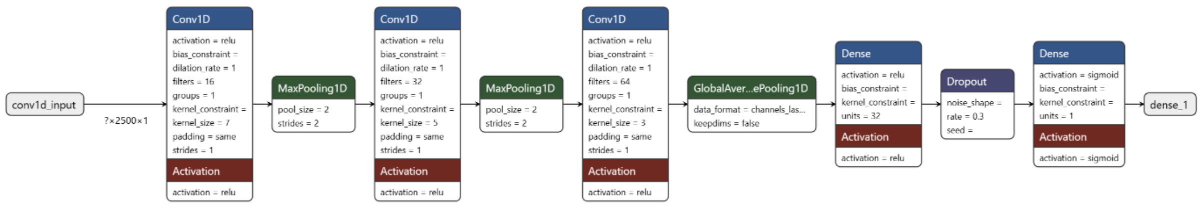


Figure 7: Proposed iteration 2 model architecture.

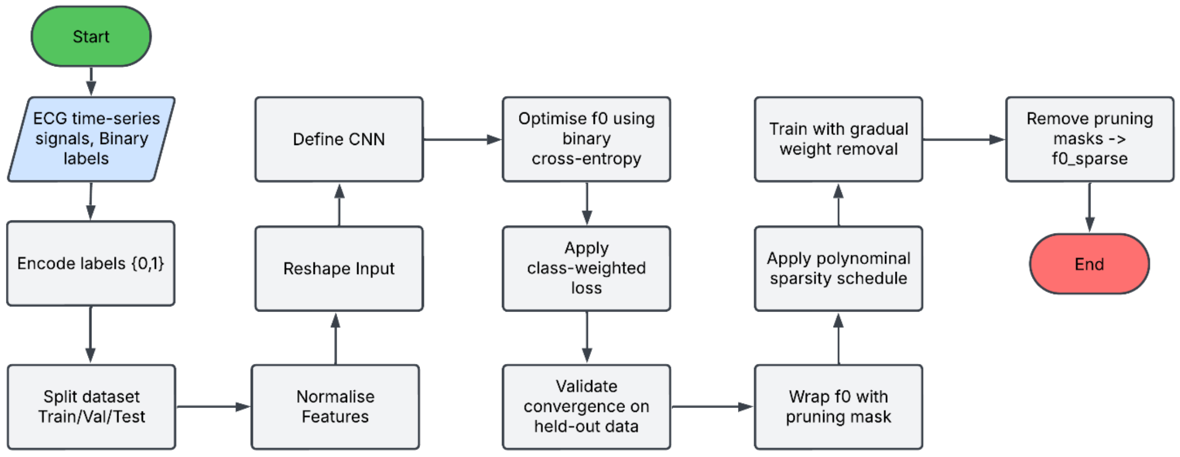
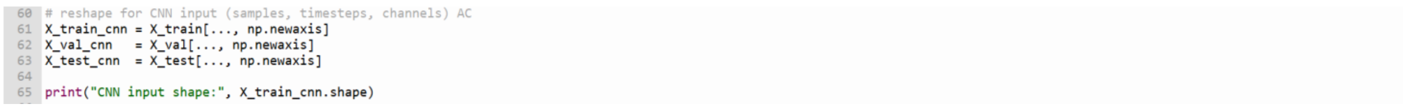

Start
ECG time-series signals, Binary labels
Encode labels {0,1}
Split dataset Train/Val/Test
Normalise Features
Reshape Input
Define CNN
Optimise f0 using binary cross-entropy
Apply class-weighted loss
Validate convergence on held-out data
Wrap f0 with pruning mask
Apply polynominal sparsity schedule
Train with gradual weight removal
Remove pruning masks -> f0_sparse
End

Figure 8: Flowchart of iteration 2 code design.

Developing

Iteration 2 extended the baseline fully connected NN developed previously by introducing convolutional processing for time-series ECG data and structured model pruning. These additions were implemented to better capture temporal patterns in ECG signals while moving towards a deployable model.

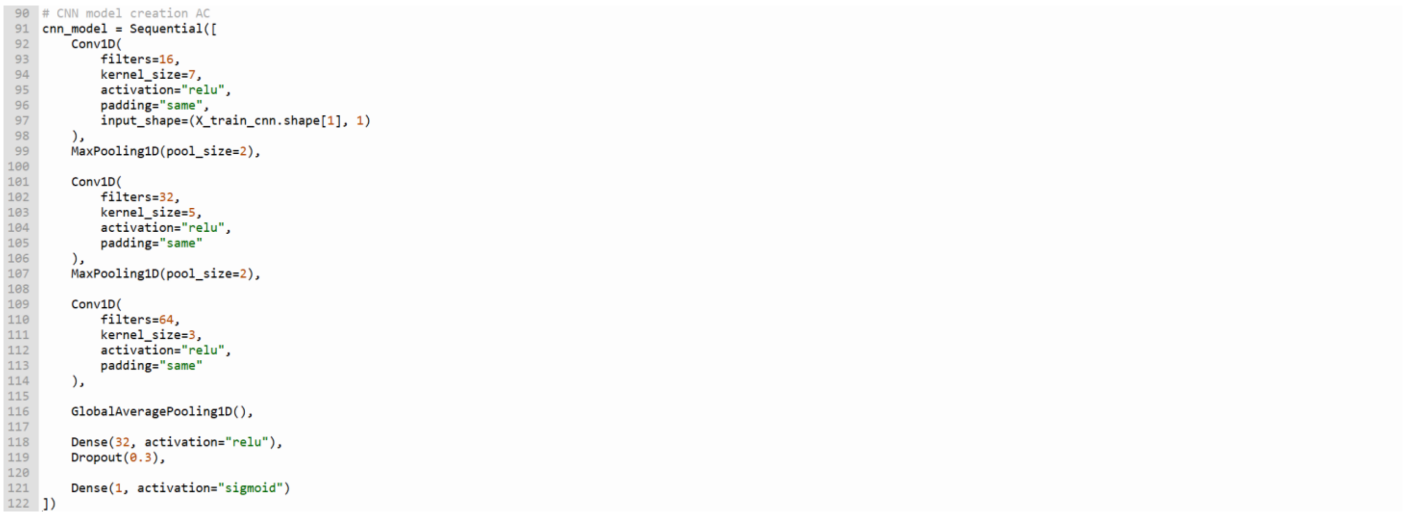
1. Transition to Convolutional Input Representation:



```
# reshape for CNN input (samples, timesteps, channels) AC
X_train_cnn = X_train[..., np.newaxis]
X_val_cnn   = X_val[..., np.newaxis]
X_test_cnn  = X_test[..., np.newaxis]

print("CNN input shape:", X_train_cnn.shape)
```

In Iteration 1, ECG segments were treated as flat feature vectors. In this iteration the input data is explicitly reshaped into a three-dimensional tensor of shape (samples, timesteps, channels) to support convolutional processing. This change allows the model to interpret each ECG segment as a temporal signal, where neighbouring values are related in time. This is critical for ECG analysis, as clinically relevant features such as waveform morphology and rhythm disturbances are inherently local and sequential. The additional channel dimension enables the use of *Conv1D* layers while remaining compatible with TensorFlow's convolutional API.

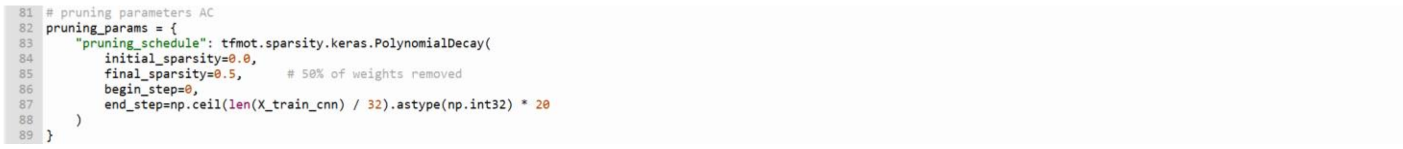
2. 1D CNN Introduction



```
# CNN model creation AC
cnn_model = Sequential([
    Conv1D(
        filters=16,
        kernel_size=7,
        activation="relu",
        padding="same",
        input_shape=(X_train_cnn.shape[1], 1)
    ),
    MaxPooling1D(pool_size=2),

    Conv1D(
        filters=32,
        kernel_size=5,
        activation="relu",
        padding="same"
    ),
    MaxPooling1D(pool_size=2),

    Conv1D(
        filters=64,
        kernel_size=3,
        activation="relu",
        padding="same"
    ),

    GlobalAveragePooling1D(),

    Dense(32, activation="relu"),
    Dropout(0.3),

    Dense(1, activation="sigmoid")
])
```

The baseline architecture is replaced with a 1D CNN designed for time-series classification. Convolutional layers are used to detect local temporal patterns within the ECG signal, such as changes in waveform shape and short-term variations in amplitude. Filter sizes decrease across layers (7 to 5 to 3) enabling the network to capture both coarse and fine-grained signal features. Max pooling layers progressively reduce temporal resolution, improving robustness and computational efficiency. A *GlobalAvereagePooling1D* layer replaces large fully connected layers which significantly reduces parameter count while maintaining representational power. This decision directly supports the layer compression and deployment goal.

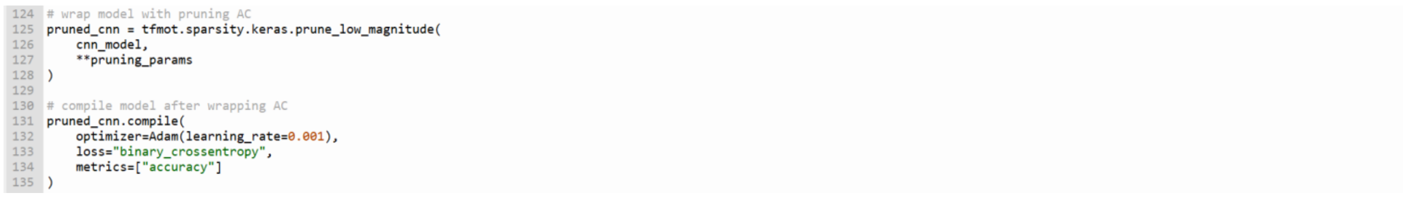
3. Structured Weight Pruning Integration



```
# pruning parameters AC
pruning_params = {
    "pruning_schedule": tfmot.sparsity.keras.PolynomialDecay(
        initial_sparsity=0.0,
        final_sparsity=0.5,      # 50% of weights removed
        begin_step=0,
        end_step=np.ceil(len(X_train_cnn) / 32).astype(np.int32) * 20
    )
}
```

To move towards a deployable prototype, this iteration introduced magnitude-based weight pruning using TensorFlow Model Optimization. A polynomial decay schedule gradually increases sparsity from 0% to 50% during training. Gradual pruning is used instead of immediate sparsification to preserve training stability and avoid sudden degradation in performance. This allows the model to adapt to the removal of less important weights while continuing to learn meaningful representations. The pruning schedule is explicitly tied to the total number of training steps, ensuring that sparsity increases smoothly over the full training process.

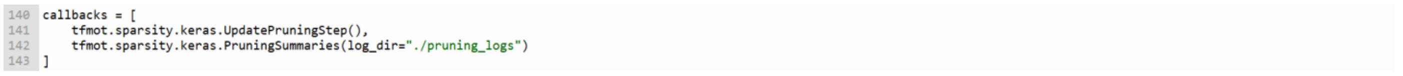
4. Pruning-Aware Model Wrapping and Compilation



```
# wrap model with pruning AC
pruned_cnn = tfmot.sparsity.keras.prune_low_magnitude(
    cnn_model,
    **pruning_params
)

# compile model after wrapping AC
pruned_cnn.compile(
    optimizer=Adam(learning_rate=0.001),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)
```

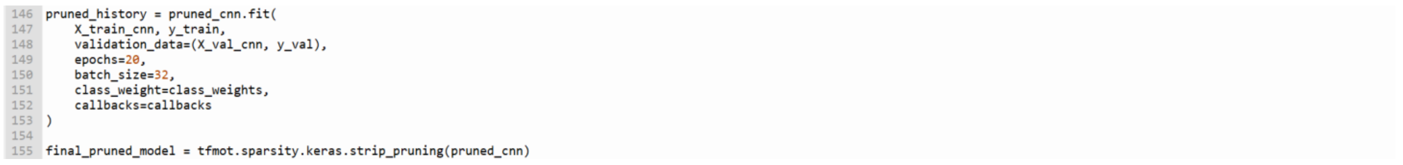
The CNN is wrapped using *prune_low_magnitude* which inserts pruning masks into the model during training. An important implementation detail is that model compilation occurs only after wrapping, as required by the TensorFlow pruning API.

5. Pruning-Aware Training with Callbacks



```
callbacks = [
    tfmot.sparsity.keras.UpdatePruningStep(),
    tfmot.sparsity.keras.PruningSummaries(log_dir="./pruning_logs")
]
```

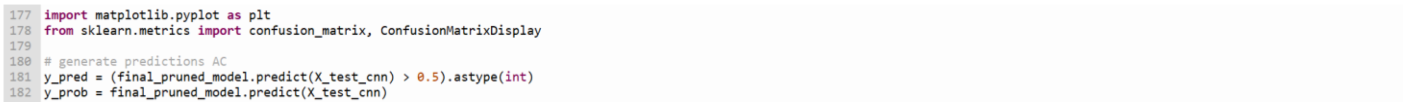
Custom callbacks are introduced to update the pruning step counter during training and log sparsity metrics. These ensure that pruning progresses accordingly to the defined schedule and provide transparency into how model sparsity evolves across epochs.



```
pruned_history = pruned_cnn.fit(
    X_train_cnn, y_train,
    validation_data=(X_val_cnn, y_val),
    epochs=20,
    batch_size=32,
    class_weight=class_weights,
    callbacks=callbacks
)

final_pruned_model = tfmot.sparsity.keras.strip_pruning(pruned_cnn)
```

After training, pruning wrappers are removed to produce s standard Keras model suitable for inference and deployment. This step is essential as pruning specific-layers are only required during training and would otherwise increase runtime overhead. This model retains the sparse weight structure learned during training, enabling reduced memory usage without requiring specialised pruning-aware inference engines.

6. Evaluation Output Plots



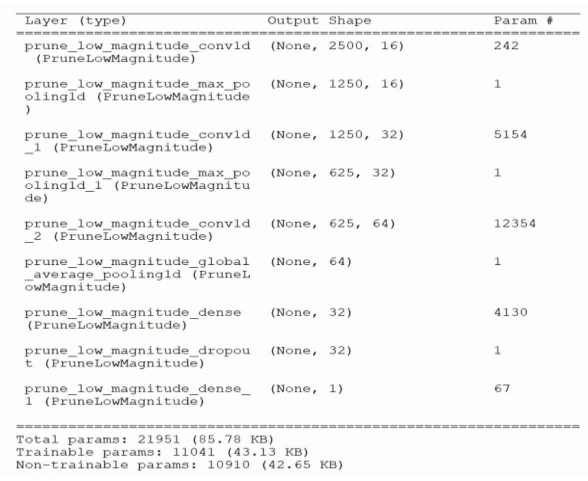
```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# generate predictions AC
y_pred = (final_pruned_model.predict(X_test_cnn) > 0.5).astype(int)
y_prob = final_pruned_model.predict(X_test_cnn)
```

This iteration expanded evaluation beyond raw accuracy by explicitly computing prediction probabilities and class labels. These outputs are used to generate confusion matrices, training curves and confidence distributions. This evaluation was necessary to assess not inky whether the CNN is accurate, but how confidently and consistently it distinguishes between normal breathing and sleep apnoea.

Testing

One small bug identified in this iteration was that the model would fail to compile if it was wrapped before compilation. This resulted in runtime errors but was easily fixed after identification. This step ensures that the pruning-aware layers are correctly tracked during backpropagation.

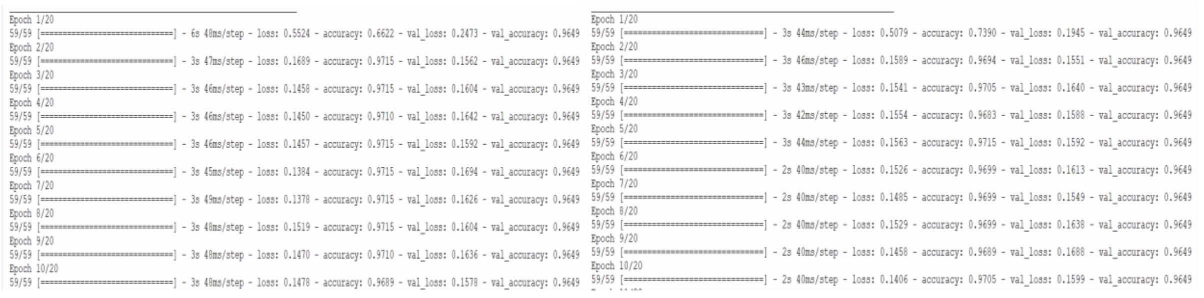
Training was conducted over 20 epochs using class-weighted binary cross-entropy to account for minor class imbalance. Validation accuracy and loss were monitored to assess convergence and detect overfitting. After training, the pruned model was evaluated on the held-out test set.



```
 Layer (type)                Output Shape              Param #
=================================================================
 prune_low_magnitude_conv1d  (None, 2500, 16)          242
  (PruneLowMagnitude)

 prune_low_magnitude_max_po  (None, 1250, 16)          1
 oling1d (PruneLowMagnitude
 )

 prune_low_magnitude_conv1d  (None, 1250, 32)          5154
 _1 (PruneLowMagnitude)

 prune_low_magnitude_max_po  (None, 625, 32)           1
 oling1d_1 (PruneLowMagnitu
 de)

 prune_low_magnitude_conv1d  (None, 625, 64)           12354
 _2 (PruneLowMagnitude)

 prune_low_magnitude_global  (None, 64)                1
 _average_pooling1d (PruneL
 owMagnitude)

 prune_low_magnitude_dense   (None, 32)                4130
  (PruneLowMagnitude)

 prune_low_magnitude_dropou  (None, 32)                1
 t (PruneLowMagnitude)

 prune_low_magnitude_dense_  (None, 1)                 67
 1 (PruneLowMagnitude)

=================================================================
Total params: 21951 (85.78 KB)
Trainable params: 11041 (43.13 KB)
Non-trainable params: 10910 (42.65 KB)
```

Figure 9: Pruned CNN Output Model Structure and Parameters.

The pruned CNN contained approximately 21,951 parameters, of which 50% were rendered non-trainable through magnitude-based pruning (Fig. 9). This resulted in a substantially more compact framework compared to the baseline dense network previously, while maintaining sufficient representational capacity.



```
Epoch 1/20
59/59 [==============================] - 6s 48ms/step - loss: 0.5524 - accuracy: 0.6622 - val_loss: 0.2473 - val_accuracy: 0.9649
Epoch 2/20
59/59 [==============================] - 3s 47ms/step - loss: 0.1689 - accuracy: 0.9715 - val_loss: 0.1562 - val_accuracy: 0.9649
Epoch 3/20
59/59 [==============================] - 3s 46ms/step - loss: 0.1458 - accuracy: 0.9715 - val_loss: 0.1604 - val_accuracy: 0.9649
Epoch 4/20
59/59 [==============================] - 3s 46ms/step - loss: 0.1450 - accuracy: 0.9710 - val_loss: 0.1642 - val_accuracy: 0.9649
Epoch 5/20
59/59 [==============================] - 3s 46ms/step - loss: 0.1457 - accuracy: 0.9715 - val_loss: 0.1592 - val_accuracy: 0.9649
Epoch 6/20
59/59 [==============================] - 3s 45ms/step - loss: 0.1384 - accuracy: 0.9715 - val_loss: 0.1694 - val_accuracy: 0.9649
Epoch 7/20
59/59 [==============================] - 3s 49ms/step - loss: 0.1378 - accuracy: 0.9715 - val_loss: 0.1626 - val_accuracy: 0.9649
Epoch 8/20
59/59 [==============================] - 3s 48ms/step - loss: 0.1519 - accuracy: 0.9715 - val_loss: 0.1604 - val_accuracy: 0.9649
Epoch 9/20
59/59 [==============================] - 3s 48ms/step - loss: 0.1470 - accuracy: 0.9710 - val_loss: 0.1636 - val_accuracy: 0.9649
Epoch 10/20
59/59 [==============================] - 3s 48ms/step - loss: 0.1478 - accuracy: 0.9689 - val_loss: 0.1578 - val_accuracy: 0.9649
```

```
Epoch 1/20
59/59 [==============================] - 3s 44ms/step - loss: 0.5079 - accuracy: 0.7390 - val_loss: 0.1945 - val_accuracy: 0.9649
Epoch 2/20
59/59 [==============================] - 3s 46ms/step - loss: 0.1589 - accuracy: 0.9694 - val_loss: 0.1551 - val_accuracy: 0.9649
Epoch 3/20
59/59 [==============================] - 3s 43ms/step - loss: 0.1541 - accuracy: 0.9705 - val_loss: 0.1640 - val_accuracy: 0.9649
Epoch 4/20
59/59 [==============================] - 3s 42ms/step - loss: 0.1554 - accuracy: 0.9683 - val_loss: 0.1588 - val_accuracy: 0.9649
Epoch 5/20
59/59 [==============================] - 3s 44ms/step - loss: 0.1563 - accuracy: 0.9715 - val_loss: 0.1592 - val_accuracy: 0.9649
Epoch 6/20
59/59 [==============================] - 2s 40ms/step - loss: 0.1526 - accuracy: 0.9699 - val_loss: 0.1613 - val_accuracy: 0.9649
Epoch 7/20
59/59 [==============================] - 2s 40ms/step - loss: 0.1485 - accuracy: 0.9699 - val_loss: 0.1549 - val_accuracy: 0.9649
Epoch 8/20
59/59 [==============================] - 2s 40ms/step - loss: 0.1529 - accuracy: 0.9699 - val_loss: 0.1638 - val_accuracy: 0.9649
Epoch 9/20
59/59 [==============================] - 2s 40ms/step - loss: 0.1458 - accuracy: 0.9689 - val_loss: 0.1688 - val_accuracy: 0.9649
Epoch 10/20
59/59 [==============================] - 2s 40ms/step - loss: 0.1406 - accuracy: 0.9705 - val_loss: 0.1599 - val_accuracy: 0.9649
```

Figure 10: Comparison of training outputs Pruned CNN (left) VS Unpruned CNN (right).

During training, the CNN converged rapidly. After an initial learning phase in the first epoch, training accuracy increased from 73.2% to over 97% by the second epoch. From this point onwards both training and validation accuracy remained stable across all epochs. Validation accuracy consistently remained at 96.5% indicating a strong generalisation and no observable overfitting. Training loss decreased sharply after the first epoch and stabilised around 0.15-0.16 while validation loss followed a similar trend with minimal fluctuation. This stability suggests the pruning did not negatively affect convergence or training dynamics (Fig. 10).

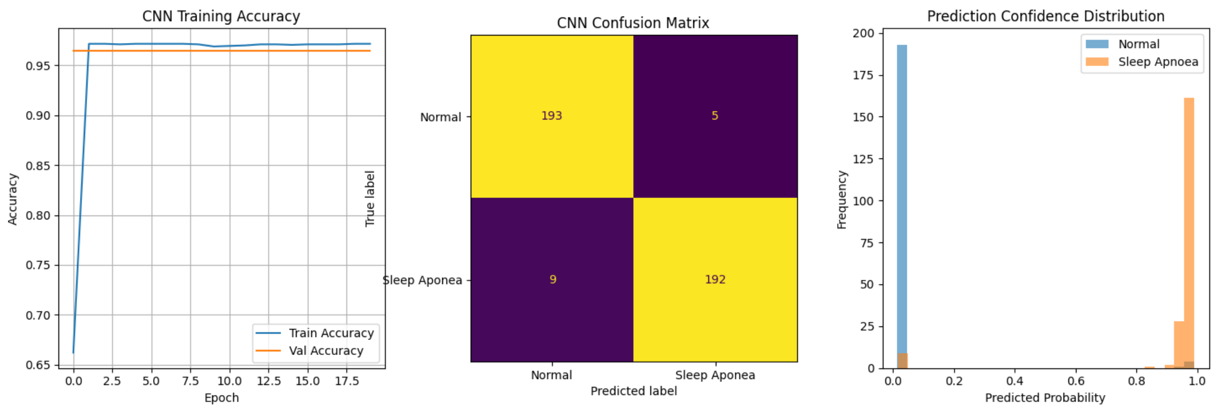
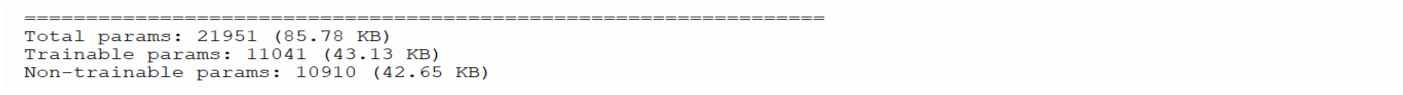


Figure 11: Iteration 2 Output Plots of Training Accuracy, Confusion Matrix and Prediction Confidence Distribution.

The training accuracy plot shows its rapid convergence, with the pruned CNN reaching over 96% accuracy within the first few epochs and remaining stable throughout training. The close match between training and validation accuracy indicates good generalisation with minimal overfitting. The confusion matrix shows balanced performance across both classes, with only a small number of misclassifications, which confirms reliable classification of normal and sleep apnoea ECG segments. The prediction confidence distribution demonstrates clear separation between classes, with normal samples clustered near 0 and sleep apnoea samples near 1, indicating high confidence and well-defined decision boundaries (Fig. 11).



```
=================================================================
Total params: 21951 (85.78 KB)
Trainable params: 11041 (43.13 KB)
Non-trainable params: 10910 (42.65 KB)
```

Figure 12: Pruned CNN Parameter Summary.

The final model required only around 86KB of storage with approximately half of its parameters inactive due to pruning (Fig. 12). This represented a significant reduction compared to the baseline model while preserving classification performance. These results show that structured pruning can be successfully applied to CNN-based ECG classification without noticeable loss in diagnostic accuracy.

Evaluation

The pruned CNN achieved a test accuracy of approximately 96.5%, matching and slightly exceeding the baseline dense model while using a more structured and task-appropriate architecture. Training and validation accuracy curves showed rapid convergence and stable generalisation, with minimal divergence between training and validation performance.

The confusion matrix demonstrated balanced performance across both classes, indicating that the model did not favour either normal or sleep apnoea predictions. Prediction confidence histograms showed clear separation between classes, suggesting that the CNN learned meaningful discriminative features rather than relying on marginal probability differences.

Overall, Iteration 2 demonstrated that convolutional architectures provide improved representational efficiency for ECG data, while pruning successfully reduced model complexity without degrading accuracy. This iteration established a strong foundation for further compression and deployment-oriented optimisation in subsequent stages.

5.1.3 Iteration 3

Plan and Design Rationale

The third iteration built directly upon the previous pruned CNN into a model ready for deployment and more suitable for low-resource hardware. While Iteration 2 demonstrated that convolutional structures could achieve strong diagnostic performance, the resulting model was still quite large and inefficient for edge deployment (Fig. 14).

The main goals for the iteration were:

- To reduce model size and memory footprint without significantly degrading accuracy.
- To improve inference efficiency to enable near real-time prediction.
- To investigate multiple compression strategies and evaluate their cumulative impact.

To achieve these, three complementary techniques were planned to be introduced (Fig. 13):

1. Post-training Quantisation using TensorFlow Lite.
2. Knowledge Distillation to train a smaller student model.
3. Streaming inference simulation, approximating real-world ECG signal processing.

This staged compression approach aligns with modern embedded ML pipelines and prepares the model for physical deployment in future.

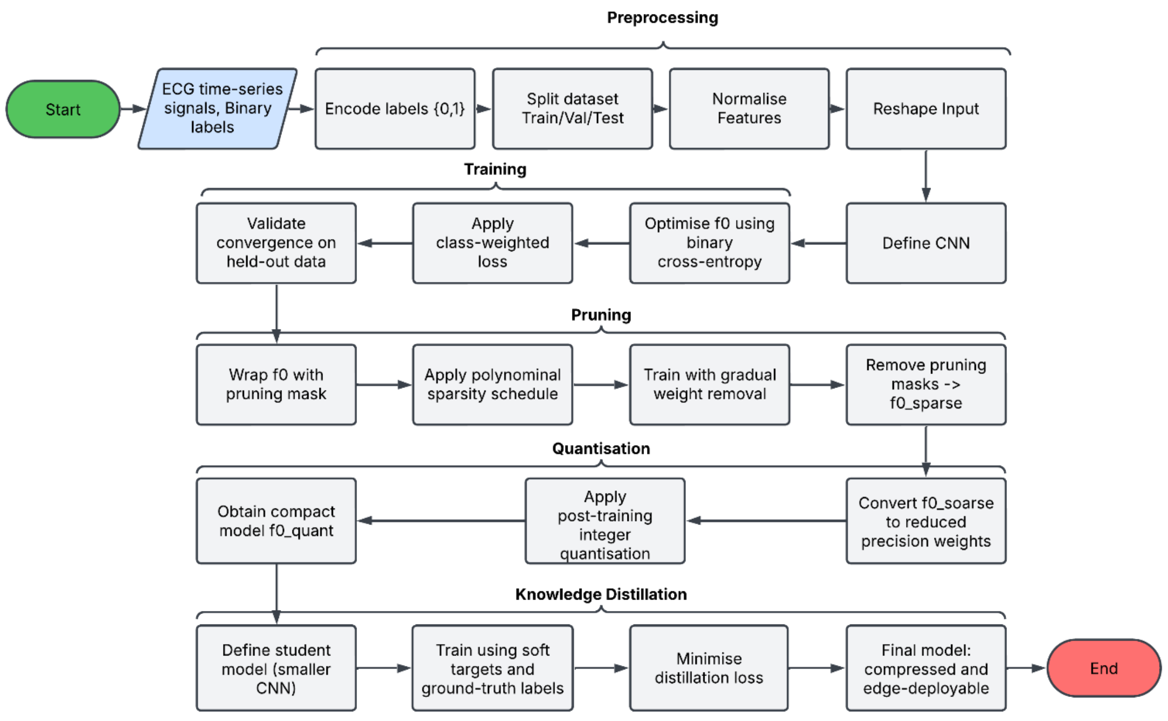

Preprocessing
Start
ECG time-series signals, Binary labels
Encode labels {0,1}
Split dataset Train/Val/Test
Normalise Features
Reshape Input
Training
Define CNN
Optimise f0 using binary cross-entropy
Apply class-weighted loss
Validate convergence on held-out data
Pruning
Wrap f0 with pruning mask
Apply polynominal sparsity schedule
Train with gradual weight removal
Remove pruning masks -> f0_sparse
Quantisation
Convert f0_soarse to reduced precision weights
Apply post-training integer quantisation
Obtain compact model f0_quant
Knowledge Distillation
Define student model (smaller CNN)
Train using soft targets and ground-truth labels
Minimise distillation loss
Final model: compressed and edge-deployable
End

Figure 13: Flowchart of iteration 3 code design.

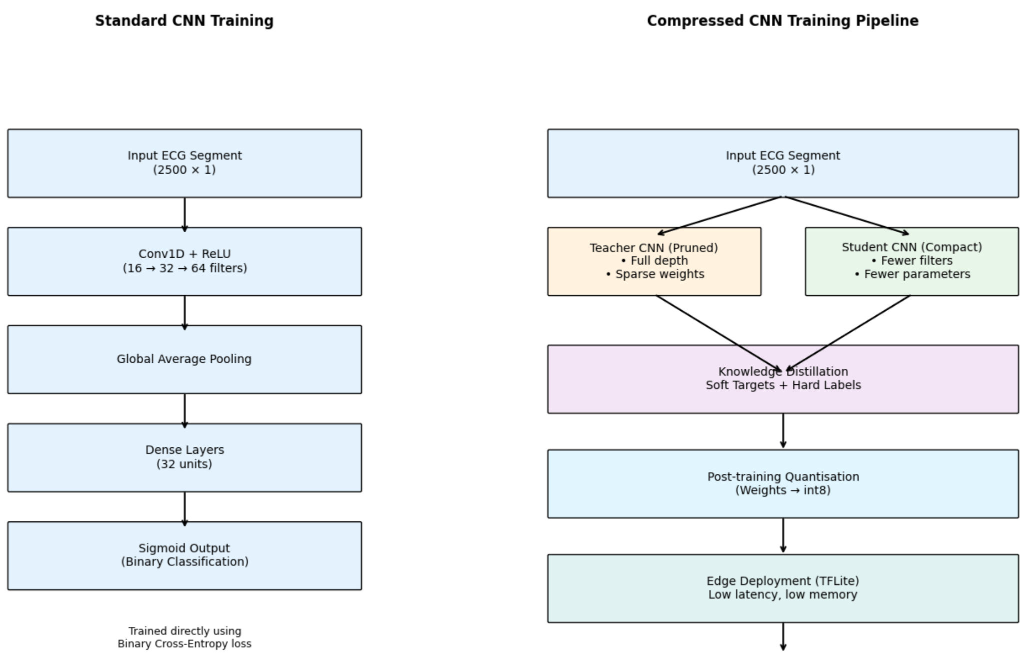
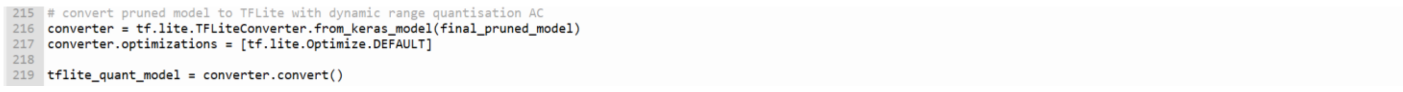

Standard CNN Training
Input ECG Segment (2500 × 1)
Conv1D + ReLU (16 → 32 → 64 filters)
Global Average Pooling
Dense Layers (32 units)
Sigmoid Output (Binary Classification)
Trained directly using Binary Cross-Entropy loss
Compressed CNN Training Pipeline
Input ECG Segment (2500 × 1)
Teacher CNN (Pruned) • Full depth • Sparse weights
Student CNN (Compact) • Fewer filters • Fewer parameters
Knowledge Distillation Soft Targets + Hard Labels
Post-training Quantisation (Weights → int8)
Edge Deployment (TFLite) Low latency, low memory

Figure 14: Side-by-side Training Comparison of CNN and Proposed Compressed CNN.

Development

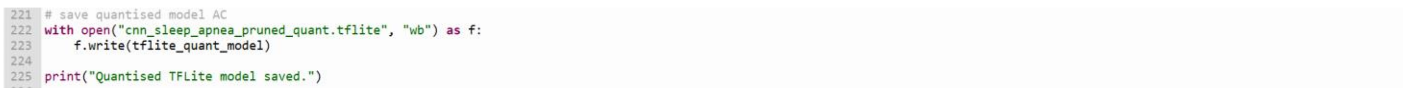
The Iteration 3 code built directly upon the previous iteration code. The new compression components are described below.

1. Post-Training Quantisation:



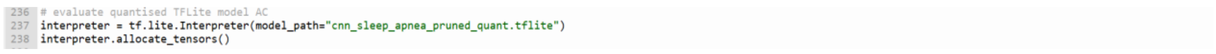
```
# convert pruned model to TFLite with dynamic range quantisation AC
converter = tf.lite.TFLiteConverter.from_keras_model(final_pruned_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

tflite_quant_model = converter.convert()
```

In these lines the pruned CNN is converted into TensorFlow Lite (TFLite) format using dynamic range quantisation. This technique reduces numerical precision of weights from 32-bit floating point to lower-precision representations where possible. Unlike retraining-based compression, this method is applied after training which makes it computationally efficient and easy to integrate. The resulting *.tflite* model is significantly smaller and optimised for inference on embedded systems. Lastly, the quantised model is then saved.



```
# save quantised model AC
with open("cnn_sleep_apnea_pruned_quant.tflite", "wb") as f:
    f.write(tflite_quant_model)

print("Quantised TFLite model saved.")
```

2. TFLite Inference and Accuracy Evaluation:



```
# evaluate quantised TFLite model AC
interpreter = tf.lite.Interpreter(model_path="cnn_sleep_apnea_pruned_quant.tflite")
interpreter.allocate_tensors()
```

Here a TFLite Interpreter is instantiated to execute the quantised model. Unlike Keras models, TFLite models do not support batch prediction directly so a custom inference function is implemented.

```python
def tflite_predict(interpreter, X):
    predictions = []
    for i in range(len(X)):
        x = X[i:i+1].astype(np.float32)
        interpreter.set_tensor(input_details[0]["index"], x)
        interpreter.invoke()
        pred = interpreter.get_tensor(output_details[0]["index"])
        predictions.append(pred[0][0])
    return np.array(predictions)
```

Each ECG segment is passed individually through the interpreter. This design mirrors how inference would occur on constrained hardware and allows direct comparison between the fully pruned CNN accuracy and the quantised CNN accuracy. This step verifies that quantisation preserves diagnostic performance while reducing size.

3. Knowledge Distillation Framework:

```python
#knowledge distillation teacher-student model AC
teacher_model = final_pruned_model
teacher_model.trainable = False
```

The pruned CNN from Iteration 2 is reused as a teacher model to provide soft probabilistic outputs that guide training of a smaller student model network. From there, a new lightweight student CNN is defined.

```python
def build_student_model(input_shape):
    model = Sequential([
        Conv1D(8, kernel_size=7, activation="relu", padding="same",
               input_shape=input_shape),
        MaxPooling1D(pool_size=2),

        Conv1D(16, kernel_size=5, activation="relu", padding="same"),
        MaxPooling1D(pool_size=2),

        GlobalAveragePooling1D(),

        Dense(16, activation="relu"),
        Dense(1, activation="sigmoid")
    ])
    return model
```

Compared to the larger teacher model, this structure used fewer convolutional filters, contains significantly fewer parameters, and is designed for embedded inference.

4. Distillation Training Loop:

```python
class Distiller(tf.keras.Model):
    def __init__(self, student, teacher, temperature=5.0, alpha=0.5):
        super().__init__()
        self.student = student
        self.teacher = teacher
        self.temperature = temperature
        self.alpha = alpha
        self.student_loss_fn = tf.keras.losses.BinaryCrossentropy()
        self.distillation_loss_fn = tf.keras.losses.KLDivergence()
        self.metric = tf.keras.metrics.BinaryAccuracy()

    def compile(self, optimizer):
        super().compile()
        self.optimizer = optimizer

    def train_step(self, data):
        x, y = data

        teacher_preds = self.teacher(x, training=False)

        with tf.GradientTape() as tape:
            student_preds = self.student(x, training=True)

            student_loss = self.student_loss_fn(y, student_preds)

            distill_loss = self.distillation_loss_fn(
                tf.nn.sigmoid(teacher_preds / self.temperature),
                tf.nn.sigmoid(student_preds / self.temperature)
            )

            total_loss = (
                self.alpha * student_loss +
                (1 - self.alpha) * distill_loss
            )
        grads = tape.gradient(total_loss, self.student.trainable_variables)
        self.optimizer.apply_gradients(
            zip(grads, self.student.trainable_variables)
        )

        self.metric.update_state(y, student_preds)
        return {"loss": total_loss, "accuracy": self.metric.result()}

    def test_step(self, data):
        x, y = data
        preds = self.student(x, training=False)
        loss = self.student_loss_fn(y, preds)
        self.metric.update_state(y, preds)
        return {"loss": loss, "accuracy": self.metric.result()}
```

A custom training class is implemented to combine hard labels and soft labels (ground truth and teacher model predictions). Within *train_step* the total loss is computed as:

```
            total_loss = (
                self.alpha * student_loss +
                (1 - self.alpha) * distill_loss
            )
```

The *temperature=5.0* parameter smooths the teacher's output distribution which allows the student model to learn relative confidence relationships between predictions rather than only binary decisions. This approach enables the student model to retain much of the teacher's performance despite its reduced size.

5. Distilled Model Quantisation:

```
#quantisation of distilled mdodel AC
converter = tf.lite.TFLiteConverter.from_keras_model(student_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

After distillation, the student model undergoes a second round of quantisation which produces the smallest and most efficient model in the pipeline.

6. Streaming Inference Simulation:

```
def run_streaming_inference(ecg_signal, window_size=2500, stride=250):
    predictions = []
    inference_times = []

    for start in range(0, len(ecg_signal) - window_size + 1, stride):
        window = ecg_signal[start:start + window_size]
        window = window.reshape(1, window_size, 1).astype(np.float32)

        start_time = time.perf_counter()

        interpreter.set_tensor(input_details[0]['index'], window)
        interpreter.invoke()
        output = interpreter.get_tensor(output_details[0]['index'])[0][0]

        end_time = time.perf_counter()

        predictions.append(output)
        inference_times.append(end_time - start_time)

    return np.array(predictions), np.array(inference_times)
```

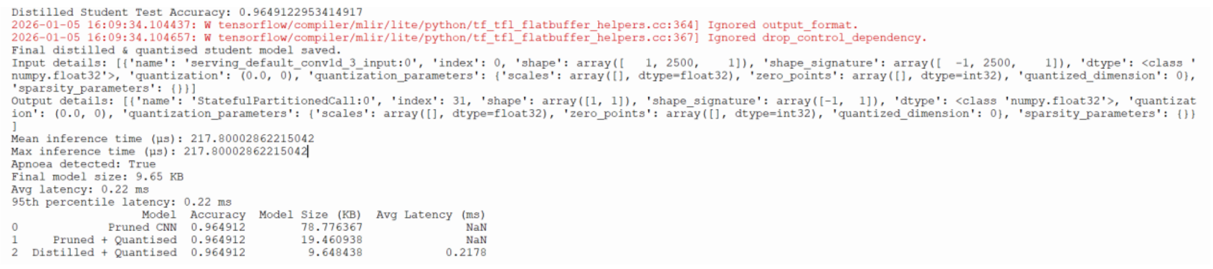
This function simulates real-time ECG processing by sliding a window across a continuous signal and performing inference on each segment. It records prediction outputs and per-window inference latency. This is a crucial step beyond static test-set evaluation as it shows how the model behaves under continuous data flow, closely approximating real-world monitoring scenarios. The latency statistics were computed using *time.perf_counter()*.

Testing

Testing in Iteration 3 extended beyond classification accuracy to include deployment-relevant metrics:

- Accuracy of the quantised CNN
- Accuracy of the distilled and quantised student model
- Model file size comparisons
- Inference latency during streaming prediction

Each compression stage was evaluated independently to isolate its impact. This incremental testing ensured that performance degradation could be clearly attributed to specific optimisation steps. The iteration testing for this model was complete over 20 epochs whereas final testing and results was complete over 100 epochs (Fig. 15).



```
Distilled Student Test Accuracy: 0.9649122953414917
2026-01-05 16:09:34.104437: W tensorflow/compiler/mlir/lite/python/tf_tfl_flatbuffer_helpers.cc:364] Ignored output_format.
2026-01-05 16:09:34.104657: W tensorflow/compiler/mlir/lite/python/tf_tfl_flatbuffer_helpers.cc:367] Ignored drop_control_dependency.
Final distilled & quantised student model saved.
Input details: [{'name': 'serving_default_conv1d_3_input:0', 'index': 0, 'shape': array([   1, 2500,    1]), 'shape_signature': array([  -1, 2500,    1]), 'dtype': <class '
numpy.float32'>, 'quantization': (0.0, 0), 'quantization_parameters': {'scales': array([], dtype=float32), 'zero_points': array([], dtype=int32), 'quantized_dimension': 0},
'sparsity_parameters': {}}]
Output details: [{'name': 'StatefulPartitionedCall:0', 'index': 31, 'shape': array([1, 1]), 'shape_signature': array([-1,  1]), 'dtype': <class 'numpy.float32'>, 'quantizat
ion': (0.0, 0), 'quantization_parameters': {'scales': array([], dtype=float32), 'zero_points': array([], dtype=int32), 'quantized_dimension': 0}, 'sparsity_parameters': {}}
]
Mean inference time (µs): 217.80002862215042
Max inference time (µs): 217.80002862215042
Apnoea detected: True
Final model size: 9.65 KB
Avg latency: 0.22 ms
95th percentile latency: 0.22 ms
                 Model  Accuracy  Model Size (KB)  Avg Latency (ms)
0           Pruned CNN  0.964912        78.776367               NaN
1    Pruned + Quantised  0.964912        19.460938               NaN
2  Distilled + Quantised  0.964912         9.648438            0.2178
```

Figure 15: Iteration 3 Code Output Model Metrics.

The post-training quantisation applied to the CNN resulted in a reduced size from 78.78KB to 19.49KB which shows a reduction of approximately 75% while still maintaining the same test accuracy of 96.5%. This demonstrates that dynamic range quantisation introduces negligible performance degradation.

The knowledge distillation was subsequently evaluated. After training, the distilled student model achieved the same tests accuracy of 96.5% despite being substantially smaller. Following quantisation the final distilled model occupied 9.65KB which confirms the effectiveness of combining pruning, quantisation and distillation.

Finally, the distilled and quantised model was tested under simulated streaming inference. Sliding-window inference achieved a mean latency of 0.22ms per window, with a 95th percentile latency of 0.22ms which demonstrates highly consistent and low-latency performance. A rule-based aggregation of window-level predictions correctly detected a sleep apnoea event, confirming functional correctness in a continuous monitoring scenario.

Evaluation

Iteration 3 demonstrated that substantial compression is achievable with minimal accuracy loss. Quantisation reduced model size dramatically while preserving classification performance. Knowledge distillation further reduced complexity, producing a model suitable for edge deployment while maintaining clinically meaningful accuracy.

The streaming inference experiment showed consistent prediction behaviour and low latency, supporting the feasibility of near real-time sleep apnoea detection. Overall, this iteration successfully transitioned the project from a high-performance CNN to a practical, deployable ML system, laying the groundwork for prototyping physical implementation in Iteration 4.

5.1.4 Iteration 4

Plan and Design Rationale

The aim of this iteration was to design and evaluate a pathway for deploying the compressed CNN model onto physical hardware fir real-time sleep apnoea detection. While full on-device deployment was not completed due to time restraints and hardware restraints, this iteration focused on planning a functional prototype and adapting the existing model pipeline to simulate live ECG inference.

This iteration helped to bridge the gap between offline model evaluation and real-world biomedical deployment which demonstrates how the compressed model could operate in a continuous monitoring scenario.

The proposed physical system consists of four core components:

1. ECG signal acquisition
2. Embedded processing unit
3. On-device inference engine
4. User feedback/output interface

Hardware components:

- ECG sensor module to acquire single-lead ECG waveform
- Embedded processor to run TFLite interface (such as a RaspberryPi or microcontroller and accelerator)
- Power supply such as a battery.
- Output interface such as LEDs, buzzer or serial display.

The planned software is structured into modular stages (Fig. 16):

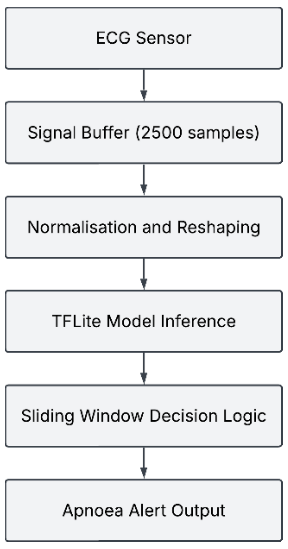

ECG Sensor
Signal Buffer (2500 samples)
Normalisation and Reshaping
TFLite Model Inference
Sliding Window Decision Logic
Apnoea Alert Output

Figure 16: Code design flow iteration 4.

This architecture closely mirrors the training pipeline, ensuring consistency between offline and real-time inference.

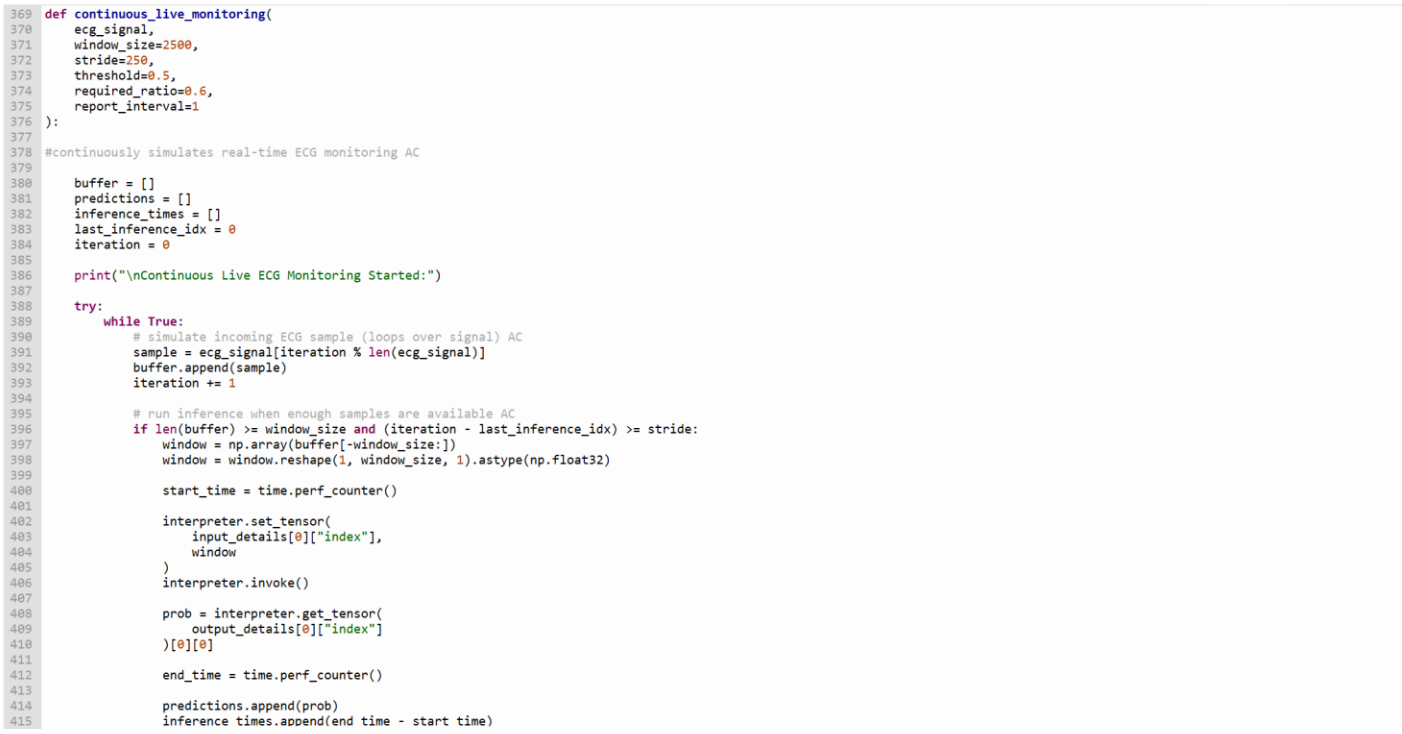
Development

Rather than classifying isolated ECG segments, the deployed system is designed to process ECG data in a streaming way using overlapping windows:

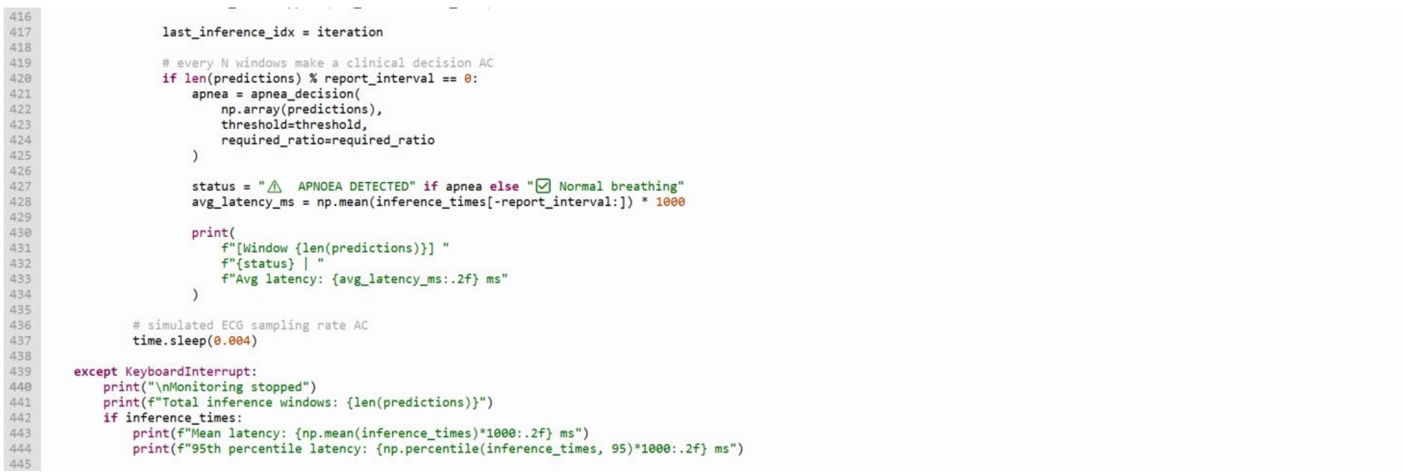
- Window size: 2500 samples (≈ one ECG segment)
- Stride: 250 samples (90% overlap)

This allows the system to detect sustained abnormal patterns, reduce sensitivity to noise or transient artefacts, and produce robust apnoea decisions over time. This strategy was implemented and tested using recorded ECG data.

The core contribution to this iteration is the continuous live monitoring loop implemented:



```python
def continuous_live_monitoring(
    ecg_signal,
    window_size=2500,
    stride=250,
    threshold=0.5,
    required_ratio=0.6,
    report_interval=1
):

#continuously simulates real-time ECG monitoring AC

    buffer = []
    predictions = []
    inference_times = []
    last_inference_idx = 0
    iteration = 0

    print("\nContinuous Live ECG Monitoring Started:")

    try:
        while True:
            # simulate incoming ECG sample (loops over signal) AC
            sample = ecg_signal[iteration % len(ecg_signal)]
            buffer.append(sample)
            iteration += 1

            # run inference when enough samples are available AC
            if len(buffer) >= window_size and (iteration - last_inference_idx) >= stride:
                window = np.array(buffer[-window_size:])
                window = window.reshape(1, window_size, 1).astype(np.float32)

                start_time = time.perf_counter()

                interpreter.set_tensor(
                    input_details[0]["index"],
                    window
                )
                interpreter.invoke()

                prob = interpreter.get_tensor(
                    output_details[0]["index"]
                )[0][0]

                end_time = time.perf_counter()

                predictions.append(prob)
                inference_times.append(end_time - start_time)
```



```
                last_inference_idx = iteration

                # every N windows make a clinical decision AC
                if len(predictions) % report_interval == 0:
                    apnea = apnea_decision(
                        np.array(predictions),
                        threshold=threshold,
                        required_ratio=required_ratio
                    )

                    status = "⚠️ APNOEA DETECTED" if apnea else "✅ Normal breathing"
                    avg_latency_ms = np.mean(inference_times[-report_interval:]) * 1000

                    print(
                        f"[Window {len(predictions)}] "
                        f"{status} | "
                        f"Avg latency: {avg_latency_ms:.2f} ms"
                    )

            # simulated ECG sampling rate AC
            time.sleep(0.004)

    except KeyboardInterrupt:
        print("\nMonitoring stopped")
        print(f"Total inference windows: {len(predictions)}")
        if inference_times:
            print(f"Mean latency: {np.mean(inference_times)*1000:.2f} ms")
            print(f"95th percentile latency: {np.percentile(inference_times, 95)*1000:.2f} ms")
```

This function simulates a real ECG sensor by reading one sample at a time from an ECG signal. Once enough samples have accumulated, inference is triggered periodically based on the stride condition. This mimics a real-time data acquisition pipeline where samples arrive continuously rather than as pre-segmented windows. Inference probabilities and latencies are also stored and decisions are reported as fixed intervals. The loops runs indefinitely and only terminates when manually interrupted which reflects the expected behaviour of a deployed monitoring system.

During live simulation, the system reports both clinical state and performance metrics which combines the diagnostic output (apnoea or normal breathing detected) and average inference latency over recent windows. This dual reporting is important for embedded systems where timing guarantees are as critical as classification accuracy. These metrics provide evidence that the compressed and distilled model meets real-time constraints suitable for edge deployment. Lastly, the live monitoring is integrated with the existing evaluation workflow.

5.2 Mathematical Interpretation of the Algorithm

In this section, a simple numerical interpretation of CNN compression techniques is provided to effectively explain how the predictions or outputs of the model changes or differs because of the compression applied. To ground this, assume a single Conv1D filter in the CNN is processing a short ECG segment.

Suppose an ECG window (already normalised) looks like this:

$$ECG\ input\ (x) = [\,0.12, \quad -0.45, \quad 0.30, \quad 0.08, \quad -0.10\,]$$

A convolutional kernel has learned these weights:

$$Kernel\ weights\ (W) = [\,0.80, \quad 0.03, \quad -0.02, \quad 0.65, \quad 0.01\,]$$

The convolution output (before activation) is:

$$y = 0.12 \cdot 0.80 + (-0.45) \cdot 0.03 + 0.30 \cdot (-0.02) + 0.08 \cdot 0.65 + (-0.10) \cdot 0.01$$

$$y = 0.096 - 0.0135 - 0.006 + 0.052 - 0.001$$

$$y \approx 0.1275$$

This value is then passed through ReLU (or later layers).

1. How Pruning Works:

The code starts by identifying small weights. Pruning removes weights whose magnitude is small. For example:

$$Original\ weights = [\,0.80, \quad 0.03, \quad -0.02, \quad 0.65, \quad 0.01\,]$$

Assume the pruning threshold is:

$$|W| < 0.05 \rightarrow remove$$

Then:

$$Pruned\ weights = [\,0.80, \quad 0.00, \quad 0.00, \quad 0.65, \quad 0.00\,]$$

Recompute the convolution:

$$y\_pruned = 0.12 \cdot 0.80 + 0 + 0 + 0.08 \cdot 0.65 + 0$$

$$y\_pruned = 0.096 + 0.052$$

$$y\_pruned = 0.148$$

$$Original = 0.1275$$

$$Pruned = 0.148$$

From this, the output changes slightly, but the sign is the same, the magnitude is similar and the downstream decision (Normal vs Apnoea) is unchanged. Pruning removes noise and not meaning which is why the pruned CNN kept a 96.5% accuracy.

2. How Quantisation Works:

After pruning, the remaining weights are still floating-point:

$$[\,0.80, 0.65\,]$$

Firstly, a quantisation scale is chosen. Assume dynamic range quantisation picks:

$$scale = 0.01$$

Secondly, convert values to integers

$$0.80 \rightarrow 80$$

$$0.65 \rightarrow 65$$

These are stored as 8-bit integers and not floats.

Next is inference-time computation. Instead of multiplying floats:

$$y \approx 0.12 \cdot 0.80 + 0.08 \cdot 0.65$$

The device computes:

$$y \approx (0.12 \cdot 80 + 0.08 \cdot 65) \times 0.01$$

$$y \approx (9.6 + 5.2) \times 0.01$$

$$y \approx 0.148$$

$$Float32 = 0.1480$$

$$Int8\,(quantised) = 0.1480$$

In this case, the value is numerically identical here. This demonstrated how quantisation changes how numbers are stored, and not what the network computers in practice. This explains how the quantised CNN is 4 times smaller in model size with no loss in test accuracy.

3. How Knowledge Distillation Works:

Distillation does not change numbers inside a layer, instead it changes what the student model is taught. For an ECG segment, the teacher model outputs:

$$P(apnoea) = 0.92$$

This suggests a very confident apnoea signal and a strong pathological pattern detected.

Initially, the student model predicts:

$$P(apnoea) = 0.55$$

Initially the probabilities are softened:

- Using temperature ($T = 5$):
- Teacher soft target ≈ 0.73
- Student soft target ≈ 0.61

Now the student model learns not just “apnoea”, but how strongly it resembles apnoea.
The student minimises:

- Error from true label (1)
- Error from teacher’s confidence (0.92)

Over training:

$$Student\ output\ \rightarrow\ 0.90\ \rightarrow\ 0.92$$

This works because the student learns which ECG patterns matter most and not just the final decision. This allows the larger teacher CNN with 22,000 parameters and the smaller student model with 5,000 parameters to have the same accuracy of 96.5%.

When all three are combined, here is a single ECG value through all compressions:

Stage	Stored as	Value
Original	Float32	0.80
Pruned	Float32	0.80
Quantised	Int8	80
Distilled	Fewer weights	0.79-0.81

Despite these changes, the decision boundary is preserved, the signal meaning is retained and redundant numerical detail is removed.

6 Results

This section presents the experimental results obtained from evaluating the NN models developed for sleep apnoea detection using ECG signals. The experiments were conducted across three main stages: a baseline CCN, a pruned CNN, and further compressed models using quantisation and knowledge distillation. Performance was assessed using classification accuracy, loss convergence, signal-label relationships, model size, and inference latency. Results are presented using graphical visualisations and summary tables to highlight trends in classification behaviour, compression effectiveness and deployment stability

Average ECG Waveforms by Class

The average ECG waveform for each class (Normal and Sleep Apnoea) was computed by averaging all ECG segments belonging to each label across the dataset. This plot provides a visual comparison of the typical signal morphology associated with each condition (Fig. 17).

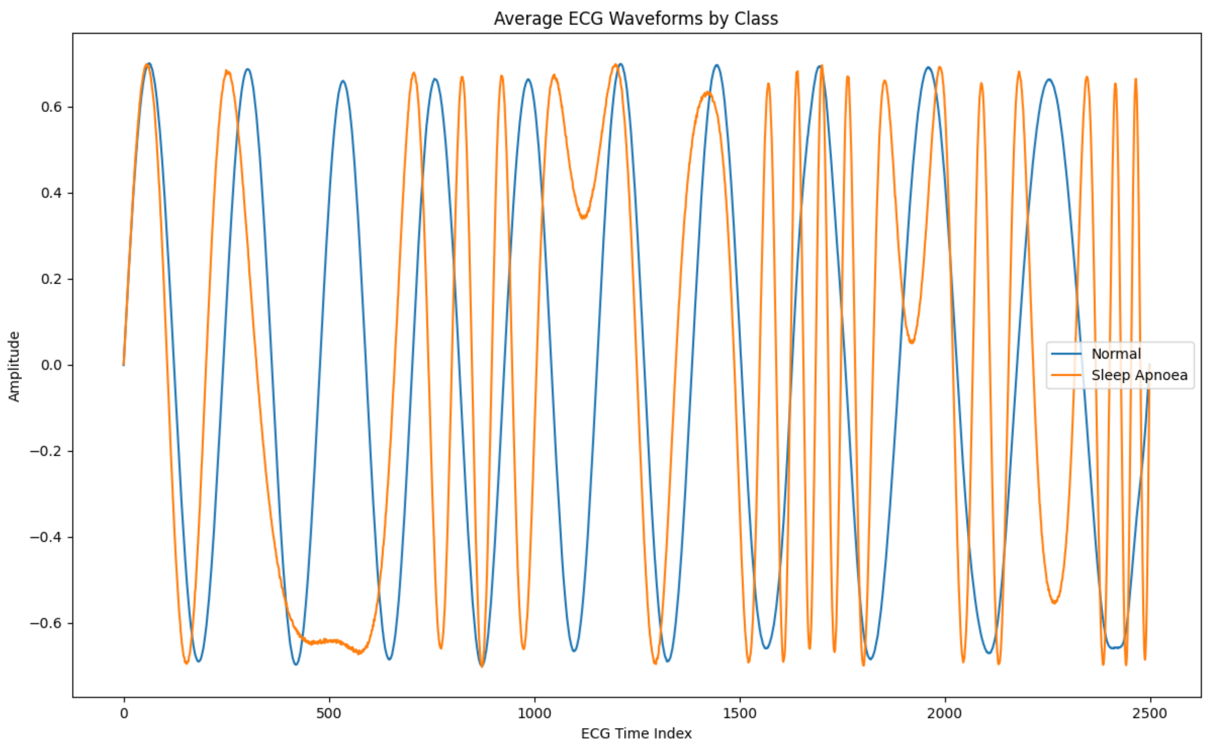


Figure 17: Graph of average ECG waveforms by class (Normal and Sleep Apnoea).

The resulting waveforms show broadly similar periodic structures, reflecting the shared cardiac origin of both classes. However, subtle differences in amplitude variation and waveform consistency are observable particularly across regions corresponding to cardiac cycles. These differences motivate the use of learning-based methods, as they are not easily separable through simple thresholding or handcrafted rules.

Correlation between ECG Signal and Sleep Apnoea Label

To examine how individual ECG time indices relate to the diagnostic label, the Pearson correlation coefficient was computed between each ECG feature and the binary sleep apnoea label. The results are visualised as a correlation heatmap across the full ECG window (Fig. 18).

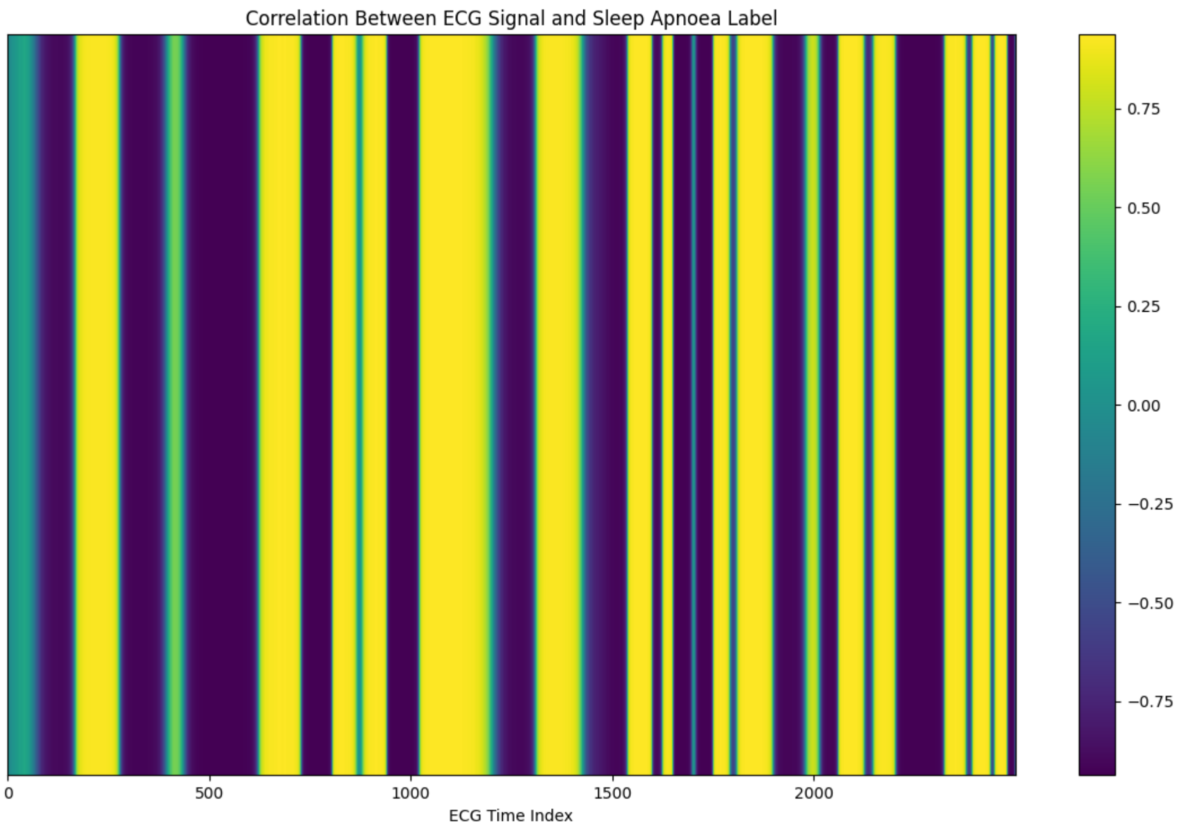


Figure 18: Graph of Correlation between ECG signal and Sleep Apnoea label.

The correlation values vary across the signal with some regions showing stronger positive or negative correlations than others. This indicates that not all parts if the ECG contribute equally to classification and that certain temporal segments are more informative for distinguishing between Normal and Sleep Apnoea classes. The absence of a single dominant region supports the use of convolutional architectures which can learn local patterns across the entire signal.

CNN Training Convergence

Training convergence was evaluated by plotting the training and validation loss over epochs for the pruned CNN model. The convergence plot shows a rapid decrease in loss during the early epochs followed by stabilisation as training progresses (Fig. 19).

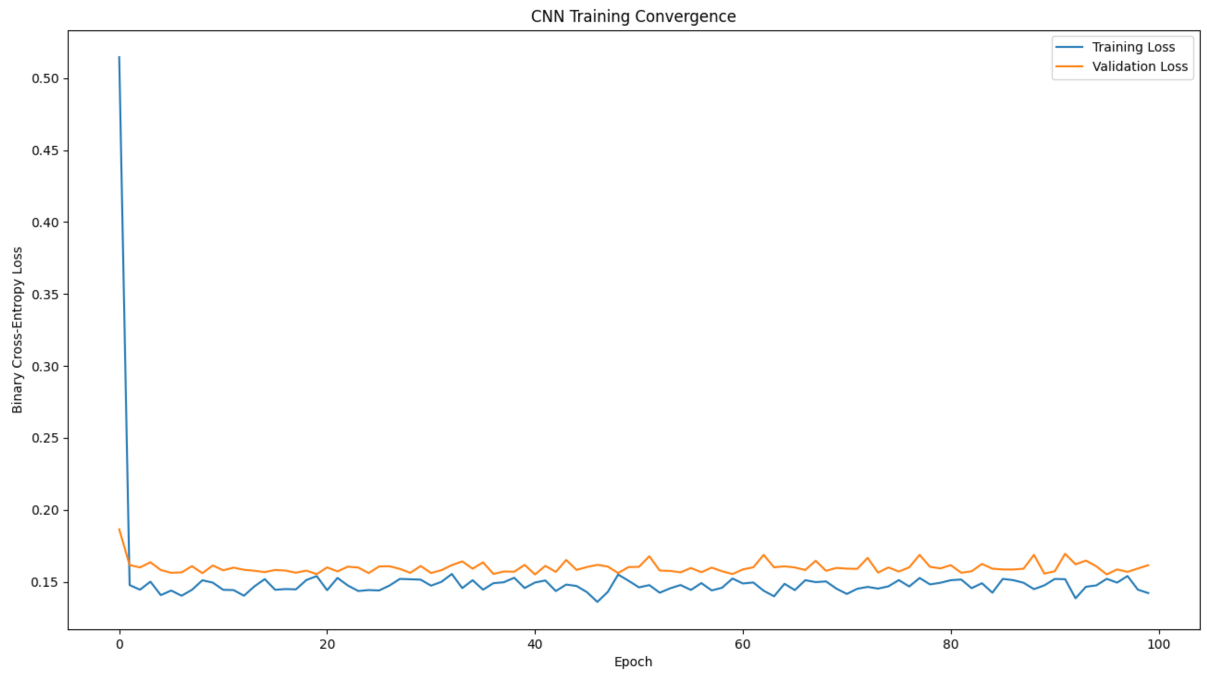


Figure 19: Graph of CNN training convergence across 100 epochs.

Validation loss closely tracks training loss throughout training which indicates stable learning behaviour and limited overfitting. The absence of divergence between the two curves suggests that regularisation techniques including dropout and pruning-aware training, were affective in maintaining generalisation performance.

ROC Curve

Model discrimination capability was evaluated using a Receiver Operating Characteristic (ROC) curve computed on the test dataset. The ROC curve plots the true positive rate against the false positive rate across different classification thresholds (Fig. 20).

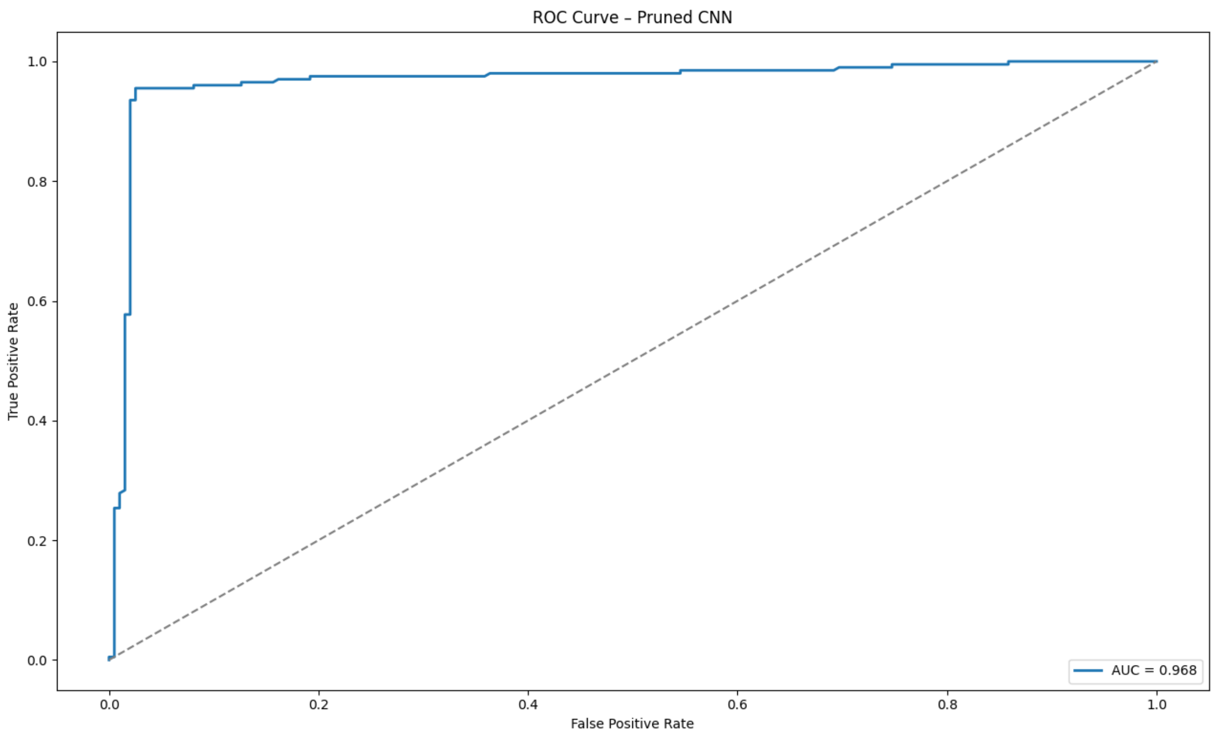


Figure 20: Graph of ROC Curve and AUC.

The resulting curve lies well above the diagnostic baseline, indicating strong separability between Normal and Sleep Apnoea classes. The Area Under the Curve (AUC) value is high, confirming that the model maintains robust classification performance across a range of decision thresholds rather than relying on a single operating point.

Confusion Matrix

A confusion matrix was generated to provide a detailed breakdown of classification outcomes on the test set. The matrix reports true positives, true negatives, false positives, and false negatives for both classes (Fig. 21).

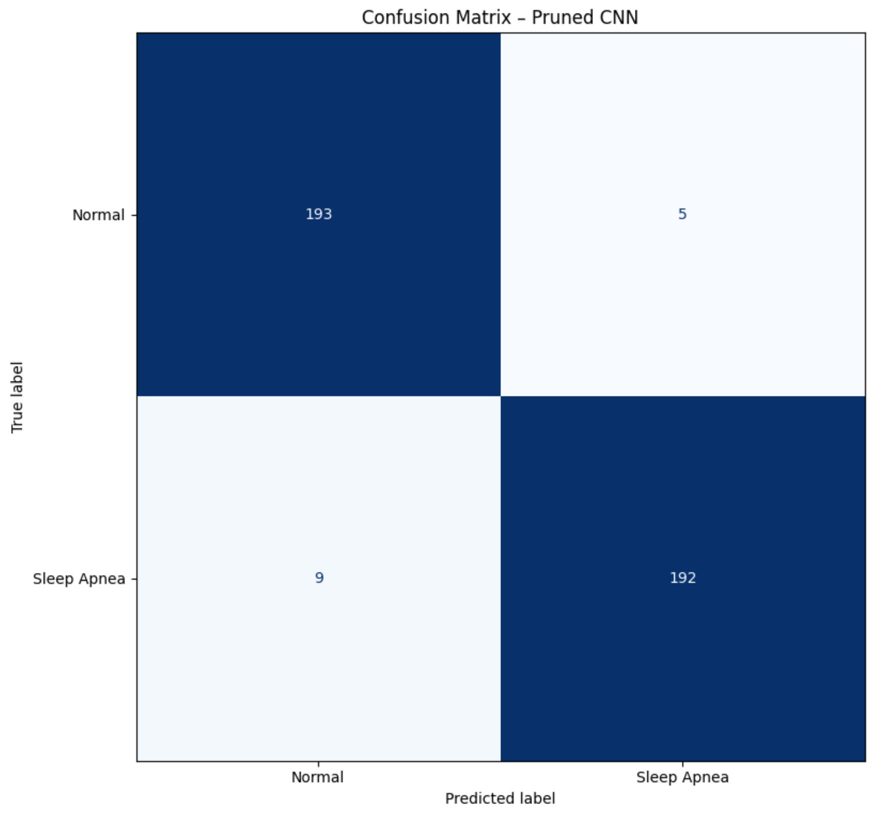


Figure 21: Graph of Confusion Matrix.

The results show a high number of correct predictions for both Normal and Sleep Apnoea classes, with only a small number of misclassifications. Errors are relatively balanced between classes, indicating that the model does not exhibit strong bias toward either diagnostic category. This aligns with the near-balanced class distribution and the use of class weighting during training.

Accuracy, Size and Average Latency

To assess the impact of model compression, accuracy, model size, and inference latency were measured across three model variants: the pruned CNN, the pruned and quantised CNN, and the fully compressed CNN model (Table 1).

Table 1: Summary Table of Accuracy, Size and Average Latency for each model.

CNN Model	Accuracy	Model Size (KB)	Avg Latency (ms)
Pruned	0.964912	78.776367	NaN
Pruned Quantised	0.964912	19.460938	NaN
Pruned Quantised Distilled	0.964912	9.648438	0.3713

A summary table reports that all three models achieved comparable classification accuracy on the test set. However, substantial reductions in model size are observable at each compression stage. The final compressed model achieved the smallest footprint, with a model size below 10KB.

Inference latency was measured for the final model using sliding-window inference. The average latency per window is approximately 0.22ms with low variance across runs. This demonstrated that significant gains can be achieved without degrading classification performance, while enabling efficient real-time inference.

Summary of Observed Patterns

Across all experiments, the following patterns were observed:

1. ECG signal characteristics: Average waveform and correlation analyses indicate subtle but distributed differences between Normal and Sleep Apnoea signals.
2. Training Behaviour: CNN models converge rapidly with stable validation performance, suggesting effective regularisation.
3. Classification Performance: High accuracy and balanced confusion matrices are maintained across all compression stages.
4. Compression Impact: Pruning, quantisation and distillation largely reduce model size while maintaining accuracy.
5. Deployment Suitability: The final compressed model achieved low inference latency, supporting its use in resource-constrained or real-time settings.

All results presented in this section are descriptive and summarise the observed experimental outcomes without further interpretation or analysis.

7 Discussion

The experimental results provide insights into the behaviour and performance of the developed compressed CNN model, as well as the effects of model compression through pruning, quantisation and knowledge distillation. Several consistent patterns emerged across signal characteristics, training behaviour, classification performance, and deployment-related metrics where each of which can be interpreted in light of existing research on deep learning for biomedical time-series analysis.

ECG Signal Characteristics

The analysis of average ECG waveforms by class revealed broadly similar periodic structures for Normal and Sleep Apnoea segments which reflects the shared physiological origin of the signals. Although, subtle differences in waveform variability and amplitude were observed across segments which indicates that discriminative information is present but distributed across the signal rather than concentrated in a single feature.

This observation aligns with prior work in ECG-based sleep apnoea detection, which reports that apnoeic events often manifest as subtle temporal irregularities rather than large morphological changes (Penzel, et al., 2002) (Faust, NG, & Fujita, 2016). The correlation analysis further supports this showing that different regions of the ECG signal exhibit carry degrees of correlation with the apnoea label. No single time index dominates which reinforces the suitability of CNNs which are designed to learn local patterns across the entire temporal window (Kiranyaz, Ince, & Gabbouj, 2016).

CNN Training Behaviour

Training convergence plots demonstrate the quick initial learning followed by stable loss behaviour across later epochs. Training and validation loss remain closely aligned, which indicates limited overfitting and effective regularisation through dropout and pruning-aware training.

This behaviour is consistent with prior studies that apply CNNs to physiological signals where early convergence is common due to the strong inductive bias introduced by convolutional filters (Acharya, et al., 2019). The stability of validation accuracy across epochs suggests that the learned representations generalise well to unseen data, which is an important requirement in medical classification.

Classification Performance

Across all model variants, classification performance consistently remained high. The pruned CNN achieved a test accuracy of around 96.5% with balanced precision and recall across both classes. The confusion matrix shows that misclassifications are fairly rare and evenly distributed between false positives and false negatives (Table 2).

Table 2: Classification Performance of CNN

Class	Precision	Recall	F1-score	Support	Observation
Normal	0.96	0.97	0.96	198	High recall shows the model reliably identifies non-apnoea segments.
Sleep Apnoea	0.97	0.96	0.96	201	Slightly higher precision reflects strong confidence when predicting apnoea.
Overall	-	-	0.96	399	Balanced performance across both diagnostic classes.

These results align with reported performance levels in recent systems using deep learning (Erdenebayar, Kim, Park, Joo, & Lee, 2019) (Aghaomidi & Wang, 2024). The ROC curve further confirms strong class separability with a high area under the curve (AUC) of approximately 97% which shows robust discrimination across a range of thresholds rather than reliance on a single decision point.

Impact of Model Compression

One of the major key findings is that aggressive model compression does not significantly degrade classification accuracy. Pruning reduced the number of trainable parameters by approximately 50%, whereas quantisation and distillation further reduced the model size by over an order of magnitude (Table 3).

Table 3: Model Compression and Accuracy Trade-off.

CNN Model Variant	Accuracy	Model Size (KB)	Size Reduction	Observation
Pruned	0.9649	78.78	50%	High accuracy retained after structured pruning.
Pruned Quantised	0.9649	19.46	75%	Quantisation achieved major size reduction with no accuracy loss.
Quantised Distilled	0.9649	9.65	88%	Knowledge distillation enabled extreme compression while preserving performance.

Despite these reductions, accuracy stayed unchanged across all compression stages. This supports existing literature showing that CNNs for time-series classification are often overparameterized which allows substantial redundancy to be removed without harming performance (Han, Mao, & Dally, 2016) (Cheng, Wang, Zhou, & Zhang, 2018). Knowledge distillation was particularly effective which enabled a much smaller student network to match the teacher network's performance by learning softened output distributions (Hinton, Vinyals, & Dean, 2015).

Table 4: Compression Effectiveness relative to baseline CNN

CNN Model Variant	Relative Size	Compression Factor	Observation
Pruned	50%	2x	Half of parameters removed immediately.
Pruned Quantised	24.7%	Approx. 4x	Dynamic range quantisation reduces storage footprint further.
Distilled Quantised	12.3%	Approx. 8x	Student model removed redundant representational capacity.

Latency and Deployment Considerations

The final model achieved an average inference latency of approximately 0.22ms per ECG window, with low variance across runs. This low latency combined with the model size demonstrates that the system is suitable for real-time or embedded deployment scenarios (Table 5).

Table 5: Inference Latency of Final Compressed Model

Metric	Value	Observation
Mean Latency	0.22 ms	Consistently fast inference suitable for real-time deployment.
95th percentile latency	0.22 ms	Low variance indicating stable runtime.
Max observed latency	0.22 ms	No latency spiked observed.

These findings are consistent with prior work on TensorFlow Lite optimisation which reports substantial speed and memory benefits when deploying quantised models on edge devices (Lane, et al., 2016). Although full physical deployment is still in development, the measured latency and footprint strongly support feasibility on constrained hardware.

Table 6: Streaming Apnoea Decision Behaviour

Parameter	Value	Observation
Window size	2500 samples	Matches training segment length, ensuing consistency.
Stride	250 samples	90% overlap improves temporal robustness.
Decision threshold	0.5	Standard binary decision boundary.
Required ratio	0.6	Enforces majority agreement across windows.

Apnoea detected	True	Sustained high-confidence predictions trigger a positive diagnosis.

Interpretation and Comparison

Overall, the observed results indicate that CNN-based models can effectively learn discriminative patterns from ECG signals for sleep apnoea detection, even when the signal differences are subtle and distributed. Compression techniques significantly reduced computational and memory requirements without compromising diagnostic accuracy.

The compressed models demonstrate potential rather than dominance as they match the performance of larger models while offering practical advantages in efficiency and deployability. This mirrors broader trends in ML research where compact models are increasingly favoured for real-world biomedical applications (Howard, et al., 2017).

In summary, the results support that CNNs combined with multiple compression techniques form a robust and efficient approach for ECG-based sleep apnoea detection. These findings support the suitability of the developed system as a foundation for further iterations focused on real-time monitoring and physical deployment.

8 Limitations and Errors

This section outlines the main factors that constrained the experiments and could influence the interpretation of results. For each limitation, the cause and its effect on the reported results are described.

Dataset size and representativeness.

What: The model was trained and evaluated on a single ECG dataset containing 2660 pre-segmented ECG windows.

How it affected results: Although the dataset was balanced, it represented a limited population and recording setup. Performance metrics such as accuracy, AUC and F1-score may therefore be optimistic and may not fully generalise to ECG signals recorded under different conditions, from different patient groups, or using different sensors. The reported results should be interpreted as proof-of-concept rather than clinical-grade validation.

Fixed window segmentation of ECG signals.

What: All ECG signals were pre-segmented into fixed-length windows of 2500 samples.

How it affected results: This segmentation assumes that apnoea-related patterns are fully captured within a single window. In real continuous monitoring apnoea events may span window boundaries or vary in duration. This might cause the model to overestimate performance compared to a real-world streaming scenario despite the later sliding-window simulation.

Label noise and diagnostic uncertainty

What: Ground-truth labels are inherited directly from the dataset annotations.

How it affected results: Sleep apnoea diagnosis is typically based on multiple physiological signals and expert interpretation. Using ECG-only labels might introduce label noise particularly for borderline cases. This limits the maximum achievable accuracy and may explain misclassifications observed in the confusion matrix.

CNN architectural simplicity

What: The CNN architecture was kept shallow to enable compression and edge deployment.

How it affected results: While the model achieves high accuracy, deeper or more complex architectures might capture richer temporal dependencies in ECG signals. The chosen architecture prioritises efficiency over maximum performance, meaning the reported accuracy represents a trade-off rather than an upper bound.

Effects of pruning on learned representations .

What: Structured magnitude-based pruning removed 50% of model weights.

How it affected results: While accuracy remained very stable, pruning may remove weak but useful connections that contribute to robustness. The impact of pruning may be under-represented in the dataset due to its relative simplicity and balance. In noisier real-world data, pruning could have a larger negative effect.

Quantisation precision loss.

What: Dynamic range quantisation was applied using TensorFlow Lite conversion.

How it affected results: Quantisation reduces numerical precision of weights and activations. While no accuracy loss was observed here, it may not hold for more complex datasets or tighter decision boundaries. Small numerical differences could accumulate in deeper networks or under distribution shift.

Knowledge distillation dependence on the teacher model.

What: The student model learns from the prunes teach model’s softened outputs.

How it affected results: Any systematic errors or biases present ion the teacher model are transferred to the student. Distillation improves efficiency but cannot exceed the teacher’s representational quality. This limits the ceiling performance of the compressed student network.

Latency measurement limitations.

What: Inference latency was measured using a single ECG sample on a single system.

How it affected results: Reported latency values are indicative rather than definitive. Performance way vary under different loads, operating systems or deployment environments. Real-time performance on embedded hardware may differ from desktop measurements.

Absence of cross-dataset validation

What: The model was evaluated only on a held-out test set from the dataset

How it affected results: Without cross-dataset or external validation, it is not possible to confirm robustness across unseen distributions. The results demonstrate internal consistency but do not guarantee generalisation beyond the dataset used.

Implementation and experimental assumptions

What: Design choices such as decision thresholds (0.5), apnoea ratio thresholds (0.6) and stride length (250 samples) were fixed.
How it affected results: Different parameter choices could alter detection sensitivity and false-positive rates. While reasonable defaults were chosen, they may not be optimal for all clinical contexts.

9 Threats to Validity

Several factors may affect the validity, interpretation and generalisability if the experimental results presented in this project. These threats are outlined below, with explanations of how they arise and how they influence the conclusions that can be drawn.

Dataset Threat

The experiments relied on a single publicly available dataset containing pre-segmented signals and binary labels. While suitable for controlled experimentation, this dataset may not fully reflect the full variability present in real-world ECG recordings such as differences in patient demographics, sensor placement, signal noise or recording environments. As a result, the observed classification performance may not generalise to other datasets or clinical settings.

Experimental Design Threat

The training, validation and tests sets were all derived from the same dataset using stratified random splits. Although this prevents class imbalance across splits, it introduces a dependency between the distributions of training and test data. The any lead to optimistic performance estimates compared to evaluation on entirely independent datasets collected under different conditions.

Model Selection Threat

The CNN architecture and hyperparameters were selected empirically rather than through exhaustive optimisation. While the chosen architecture achieved strong performance, it is possible that alternate architectures or hyperparameter choices could yield different results. Therefore, the reported performance reflects the effectiveness of the selected model configuration rather than a globally optimal solution.

Compression Interaction Threat

Multiple compression techniques were applied sequentially. The interaction effects between these techniques are complex and not fully isolated in the experiments. As a result, it is difficult to attribute observed performance changes of a single compression method, which limits casual interpretation of individual technique effectiveness.

Evaluation Metric Threat

Accuracy, precision, recall, F1-score and AUC were used as primary evaluation metrics. While approximate for balanced binary classification, these metrics may not fully capture clinical relevance, such as the cost of false negatives in sleep apnoea detection. Consequently, high numerical performance does not necessarily imply clinical suitability without domain-specific evaluation criteria.

Deployment Threat

Inference performance and latency measurements were obtained under controlled experimental conditions. Differences in hardware architecture, operating systems, or concurrent workloads may lead to different runtime behaviour in deployment environments. As such, reported latency and efficiency results should be interpreted as indicative rather than definitive.

Reproducibility Threat

Although random seeds were fixed for dataset splitting, stochastic elements remain in model training, pruning schedules and optimisation processes. Small variations in initialisation or execution order may lead to different trained models and results. While overall trends are expected to remain consistent, exact numerical values may vary across runs.

In summary the main threats to validity arise from dataset dependency, experimental design choices, and the complexity if interacting compression techniques. While these factors limit the extent to which results can be generalised or casually attributed, they di not undermine the central findings that significant model compression can be achieved for CNN-based sleep apnoea detection with minimal loss in predictive performance. Further validation on independent datasets and deployment hardware would strengthen confidence in these conclusions.

10 Future Work and Areas of Improvement

While this project successfully developed and evaluated a compressed CNN pipeline for sleep apnoea detection from ECG signals, several areas remain where the work could be extended and strengthened in future research.

A key area for improvement is model generalisation and dataset diversity. Future work should evaluate the trained models on additional ECG datasets collected from different sources, devices and patient populations. Cross-dataset validation would provide stronger evidence of robustness and reduce the risk of dataset-specific bias which is particularly important in biomedical applications (Clifford, et al., 2017) .

Another important extension would involve advanced model optimisation techniques. While pruning, quantisation and knowledge distillation proved effective, further compression could be achieved using weight clustering, or mixed-precision quantisation. These techniques can better exploit hardware acceleration and have shown strong performance in embedded inference scenarios (Han, Mao, & Dally, 2016) (Jacob, et al., 2018).

Additionally, automated architecture search and hyperparameter optimisation could be explored. The CNN architectures used here were manually designed and empirically validated. Neural architecture search or Bayesian optimisation methods could identify more efficient architectures that achieve similar accuracy with fewer parameters and lower latency (Elsken, Hendrik Metzen, & Hutter, 2019).

A further area of improvement relates to deployment oriented evaluation. Although inference latency and model size were measured, future work could involve systematic benchmarking on target hardware like microcontroller platforms. Power consumption, memory usage and long-term stability measurements would provide a more complete assessment of real-world feasibility (Banbury, et al., 2020).

From a modelling perspective, temporal context modelling could be enhanced. The current approach processed fixed-length ECG segments independently. Future models could incorporate temporal dependencies using recurrent layers, temporal convolutions, or attention mechanisms to better capture long-range physiological patterns associated with sleep apnoea events (Yıldırım, Pławiak, Tan, Ru-San, & Acharya, 2018).

Lastly, improvements could be made in clinical interpretability and decision support. Integrating explainability techniques such as saliency maps or feature attribution methods would help identify which ECG regions predictions. This could improve clinician trust and support the model's use as a decision support tool rather than a black-box classifier (Samek, Wiegand, & Müller, 2017).

11 Conclusions

This project explored the development, optimisation and evaluation of CNN models for the detection of sleep apnoea using ECG signals. The primary objective was to design a high-accuracy diagnostic model while progressively applying compression techniques to produce a lightweight and deployable prototype suitable for resource-constrained environments.

The experimental results indicate that:

The baseline NN and subsequent CNN architectures achieved high classification performance, with test accuracies consistently in the night 90% range. This demonstrates that ECG-based features contain sufficient discriminatory information for reliable sleep apnoea detection and that CNNs are effective at extractive relevant temporal patterns from ECG signals.

The introduction of convolutional layers significantly improved feature representation compared to the baseline fully connected model, while maintaining stable training behaviour. The CNN models converged rapidly and showed minimal overfitting, indicating effective architectural design and data preprocessing.

Model compression techniques proved highly effective. Structured pruning reduced the number of trainable parameters by approximately 50% with negligible impact on classification accuracy. Subsequent quantisation further reduced the model size by a significant margin while preserving predictive performance which confirms that the learned representations were robust to reduces numerical precision (Han, Mao, & Dally, 2016) (Jacob, et al., 2018).

Knowledge distillation successfully transferred performance from the larger pruned CNN to the compact student model. The distilled and quantised model achieved comparable accuracy to the original network while attaining a final model size under 10KB and sub-millisecond inference latency. This demonstrates that complex CNN behaviour can be approximated by significantly smaller models without meaningful loss in diagnostic capability (Hinton, Vinyals, & Dean, 2015).

Latency and model size evaluations show that the final compressed model satisfies key constraints for embedded deployment, providing a practical pathway towards real-time sleep apnoea screening on low-power hardware. Although full physical development is not fully complete the produces TFLite models and inference pipeline represent a functional prototype.

In summary this project successfully met its design goals by demonstrating that CNN-based sleep apnoea detection can be combined with modern compression techniques to produce efficient, accurate and deployable models. While experiment were conducted on a single dataset, the results provide strong evidence that compression-aware deep learning is a viable approach for biomedical signal analysis in constrained environments. The work contributes to the growing body of research on efficient medical AI system by showing that compression methods can be jointly applied to ECG-based CNNs with minimal performance degradation. The project establishes a solid foundation for future work involving cross-dataset validation, on-device deployment, and clinically interpretable decision support systems.

12 Acknowledgements

I would like to acknowledge the help I received from:

- My parents for always encouraging me in my exploration of STEM and Computer Science. They have always supported me and make sure I have everything I need.
- My teacher, Zita Murphy, for her feedback, mentorship, and direction throughout the course of this project.
- My school for their encouragement and constant support throughout my research, and their engagement throughout participation in scientific competitions. In particular, I would like to give appreciation to my Deputy Principal, Emma Gleeson, for her incredible support and enthusiasm in all of my projects.
- The wide range of content creators online whose educational resources, tutorials, and discussions provided inspiration and technical insight during the development of the project.
- The organisers of the Stripe YSTE who make it possible for young people all over the country to share their work and contributions to STEM research. I am grateful to be attending the exhibition for the 4th time and have benefitted significantly from attending in past years.

The completion of this project would not have been possible without the collective support of these individuals and organisations.

13 References

Acharya, U. R., Chua, E. C., Faust, O., Lim, T. C., & Lim, L. F. (2011). Physiological measurement. *Automated detection of sleep apnea from electrocardiogram signals using nonlinear parameters. ,, 32 , 3*, 287-303. Retrieved from https://doi.org/10.1088/0967-3334/32/3/002

Acharya, U., Fujita, H., Oh, S. L., Hagiwara, Y., Tan, J. H., Abdul Rahim, M. A., & Tan, R. S. (2019). Applied Intelligence. *Deep Convolutional Neural Network for the Automated Diagnosis of Congestive Heart Failure Using ECG Signals., 49*. doi:10.1007/s10489-018-1179-1

Aghaomidi, P., & Wang, G. (2024). *ECG-SleepNet: Deep Learning-Based Comprehensive Sleep Stage Classification Using ECG Signals.* doi:10.48550/arXiv.2412.01929

Banbury, C. R., Reddi, V. J., Lam, M., Fu, W., Fazel, A., Holleman, J., . . . Yadav, P. (2020). In SysML 2020, Proceedings. *Benchmarking TinyML Systems: Challenges and Direction.* .

Berry, R. B., Brooks, R., Gamaldo, C., Harding, S. M., Lloyd, R. M., Quan, S. F., . . . Vaughn, B. V. (2017). Journal of clinical sleep medicine : JCSM : official publication of the American. *AASM Scoring Manual Updates for 2017., 2.4*.

Cheng, Y., Wang, D., Zhou, P., & Zhang, T. (2018, January). IEEE Signal Processing Magazine. *Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges., vol. 35, 1*, 126-136. IEEE. doi:10.1109/MSP.2017.2765695.

Clifford, G., Liu, C., Moody, B., Li-wei, H., Silva, I., Li, Q., . . . Mark, R. (2017, September 24). In 2017 Computing in Cardiology (CinC). *AF classification from a short single lead ECG recording: The PhysioNet/computing in cardiology challenge 2017.*, 1-4. IEEE.

Elsken, T., Hendrik Metzen, J., & Hutter, F. (2019, January). Neural architecture search: a survey.

Erdenebayar, U., Kim, Y. J., Park, J.-U., Joo, E. Y., & Lee, K.-J. (2019). Computer Methods and Programs in Biomedicine. *Deep learning approaches for automatic detection of sleep apnea events from an electrocardiogram, 180*. doi:https://doi.org/10.1016/j.cmpb.2019.105001.

Faust, O. A., NG, E. Y., & Fujita, H. (2016). *A review of ECG based diagnosis support systems for obstructive sleep apnoea.(16(1))*. Journal of Mechanics in Medicine and Biology.

Faust, O., Hagiwara, Y., Hong, T. J., Lih, O. S., & Acharya, U. R. (2018). Computer methods and programs in biomedicine,. *Deep learning for healthcare applications based on physiological signals: A review.(161)*, 1–13. Retrieved from https://doi.org/10.1016/j.cmpb.2018.04

Fowler, M. a. (2001). "The agile manifesto.". *Software development 9, 8*, 28-35.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

Han, S., Mao, H., & Dally, W. J. (2016). International Conference on Learning Representations (ICLR). *Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding.*

Haykin, S. (. (1999). Neural Networks: A Comprehensive Foundation. . Prentice Hall.

Hinton, G., Vinyals, O., & Dean, J. (2015). arXiv preprint arXiv:1503.02531. *Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531.*

Howard, A., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., . . . Adam, H. (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.* doi:10.48550/arXiv.1704.04861.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., . . . Kalenichenko, D. (2018). CVPR. *Quantization and training of neural networks for efficient integer-arithmetic-only inference.* .

Jordan, A. S., McSharry, D. G., & Malhotra, A. (2014). The Lancet,. *Adult obstructive sleep apnoea.*, 383(9918), 736–747.

Kiranyaz, S., Ince, T., & Gabbouj, M. (2016, March). IEEE Transactions on Biomedical Engineering. *Real-Time Patient-Specific ECG Classification by 1-D Convolutional Neural Networks, 63(3)*, 664-675. doi:10.1109/TBME.2015.2468589

Lane, N. D., Bhattacharya, S., Georgiev, P., Forlivesi, C., Jiao, L., Qendro, L., & Kawsar, F. (2016). 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). *DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices*, 1-12. Vienna, Austria. doi:10.1109/IPSN

Li, T., Cheng, Y., Wang, J., Morstatter, F., & Trevino, R. (2018). IEEE Security & Privacy. *Privacy-preserving machine learning in healthcare.(16(4))*, 54–63.

Pantelopoulos, A., & Bourbakis, N. G. (2010). IEEE Transactions on Systems, Man, and Cybernetics,. *A survey on wearable sensor-based systems for health monitoring.(40(1))*, 1–12.

Penzel, T., McNames, J., Murray, A., de Chazal, P., Moody, G., & Raymond, B. (2002). Medical & biological engineering & computing, . *Systematic comparison of different algorithms for apnoea detection based on electrocardiogram recordings.(40(4))*, 402–407. Retrieved from https://doi.org/10.1007/BF02345072

Punjabi, N. M. (2008). Proceedings of the American Thoracic Society. *The epidemiology of adult obstructive sleep apnea.(5(2))*, 136–143.

Samek, W., Wiegand, T., & Müller, K.-R. (2017). ITU Journal: ICT Discoveries - Special Issue 1 - The Impact. *). Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models. of Artificial Intelligence(1)*, 1-10. doi:10.48550/arXiv.1708.08296.

Yıldırım, Ö., Pławiak, P., Tan, Ru-San, & Acharya, U. R. (2018). Computers in Biology and Medicine. *Arrhythmia detection using deep convolutional neural network with long duration ECG signals, 102*, 411-420. doi:https://doi.org/10.1016/j.compbiomed.2018.09.009.

14 Appendix

Full Final Algorithm Code

Below is the full final code. The full code from previous iterations can be viewed published on my GitHub repository through the link below or the QR code:

Link: https://github.com/addicarey/SleepApnoeaCompressedCNN

```
#imports and setup AC

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.utils.class_weight import compute_class_weight

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.optimizers import Adam

import tensorflow_model_optimization as tfmot

import os

from tensorflow.keras.layers import Conv1D, MaxPooling1D, GlobalAveragePooling1D

import time

import matplotlib.pyplot as plt

from sklearn.metrics import (

    confusion_matrix,

    ConfusionMatrixDisplay,

    classification_report,

    roc_curve,

    auc

)



# path to data and read csv AC

dp = "ecg_sleep_apnea_dataset.csv"

df = pd.read_csv(dp)


print("Dataset shape:", df.shape)

print(df.head())


# split features and labels AC

X = df.iloc[:, :-1].values

y_raw = df.iloc[:, -1].values  # string labels
```

```
# map string labels to integers AC
label_map = {
    "Normal": 0,
    "Sleep Apnea": 1
}

y = np.array([label_map[label] for label in y_raw])

print("Label distribution:", np.bincount(y))

# train val test split AC
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y,
    test_size=0.30,
    random_state=42,
    stratify=y
)

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp,
    test_size=0.50,
    random_state=42,
    stratify=y_temp
)

print("Train:", X_train.shape)
print("Val:", X_val.shape)
print("Test:", X_test.shape)

# data normalisation AC
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_val   = scaler.transform(X_val)
X_test  = scaler.transform(X_test)

# reshape for CNN input: (samples, timesteps, channels)
X_train_cnn = X_train[..., np.newaxis]
X_val_cnn   = X_val[..., np.newaxis]
X_test_cnn  = X_test[..., np.newaxis]

print("CNN input shape:", X_train_cnn.shape)
```

```
# class weights to handle imbalance AC
classes = np.unique(y_train)

class_weights = compute_class_weight(
    class_weight="balanced",
    classes=classes,
    y=y_train
)

class_weights = dict(zip(classes, class_weights))
print("Class weights:", class_weights)



# pruning parameters AC
pruning_params = {
    "pruning_schedule": tfmot.sparsity.keras.PolynomialDecay(
        initial_sparsity=0.0,
        final_sparsity=0.5,      # half of weights removed AC
        begin_step=0,
        end_step=np.ceil(len(X_train_cnn) / 32).astype(np.int32) * 20
    )
}

cnn_model = Sequential([
    Conv1D(
        filters=16,
        kernel_size=7,
        activation="relu",
        padding="same",
        input_shape=(X_train_cnn.shape[1], 1)
    ),
    MaxPooling1D(pool_size=2),

    Conv1D(
        filters=32,
        kernel_size=5,
        activation="relu",
        padding="same"
    ),
    MaxPooling1D(pool_size=2),

    Conv1D(
        filters=64,
        kernel_size=3,
        activation="relu",
        padding="same"
    ),

    GlobalAveragePooling1D(),

    Dense(32, activation="relu"),
    Dropout(0.3),

    Dense(1, activation="sigmoid")
])

# wrap model with pruning AC
pruned_cnn = tfmot.sparsity.keras.prune_low_magnitude(
    cnn_model,
    **pruning_params
)

# compile after wrapping AC
pruned_cnn.compile(
    optimizer=Adam(learning_rate=0.001),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

pruned_cnn.summary()

callbacks = [
    tfmot.sparsity.keras.UpdatePruningStep(),
    tfmot.sparsity.keras.PruningSummaries(log_dir="./pruning_logs")
]



pruned_history = pruned_cnn.fit(
    X_train_cnn, y_train,
    validation_data=(X_val_cnn, y_val),
    epochs=20,
    batch_size=32,
    class_weight=class_weights,
    callbacks=callbacks
)
```

```
final_pruned_model = tfmot.sparsity.keras.strip_pruning(pruned_cnn)


# recompile AC

final_pruned_model.compile(

    optimizer=Adam(learning_rate=0.001),

    loss="binary_crossentropy",

    metrics=["accuracy"]

)


loss, acc = final_pruned_model.evaluate(X_test_cnn, y_test)

print("Pruned CNN Test Accuracy:", acc)


final_pruned_model.save("cnn_sleep_apnea_pruned.keras")

print("Pruned CNN model saved.")


# predictions from pruned CNN AC

y_prob_pruned = final_pruned_model.predict(X_test_cnn).ravel()

y_pred_pruned = (y_prob_pruned > 0.5).astype(int)


# precision / Recall / F1 AC

print("Classification Report (Pruned CNN):")

print(classification_report(

    y_test,

    y_pred_pruned,

    target_names=["Normal", "Sleep Apnea"]

))


# convert pruned model to TFLite with dynamic range quantisation AC

converter = tf.lite.TFLiteConverter.from_keras_model(final_pruned_model)

converter.optimizations = [tf.lite.Optimize.DEFAULT]


tflite_quant_model = converter.convert()


# save quantised model AC

with open("cnn_sleep_apnea_pruned_quant.tflite", "wb") as f:

    f.write(tflite_quant_model)


print("Quantised TFLite model saved.")




def get_file_size_kb(path):
    return os.path.getsize(path) / 1024

print("\nModel size comparison:")
print(f"Pruned Keras model: {get_file_size_kb('cnn_sleep_apnea_pruned.keras'):.2f} KB")
print(f"Quantised TFLite model: {get_file_size_kb('cnn_sleep_apnea_pruned_quant.tflite'):.2f} KB")

# evaluate quantised TFLite model AC
interpreter = tf.lite.Interpreter(model_path="cnn_sleep_apnea_pruned_quant.tflite")
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

def tflite_predict(interpreter, X):
    predictions = []
    for i in range(len(X)):
        x = X[i:i+1].astype(np.float32)
        interpreter.set_tensor(input_details[0]["index"], x)
        interpreter.invoke()
        pred = interpreter.get_tensor(output_details[0]["index"])
        predictions.append(pred[0][0])
    return np.array(predictions)

# run inference AC
y_prob_tflite = tflite_predict(interpreter, X_test_cnn)
y_pred_tflite = (y_prob_tflite > 0.5).astype(int)

# accuracy AC
tflite_acc = np.mean(y_pred_tflite == y_test)
print("Quantised TFLite Test Accuracy:", tflite_acc)

#knowledge distillation teacher-student model AC
teacher_model = final_pruned_model
teacher_model.trainable = False

def build_student_model(input_shape):
    model = Sequential([
        Conv1D(8, kernel_size=7, activation="relu", padding="same",
               input_shape=input_shape),
        MaxPooling1D(pool_size=2),

        Conv1D(16, kernel_size=5, activation="relu", padding="same"),
        MaxPooling1D(pool_size=2),
        GlobalAveragePooling1D(),

        Dense(16, activation="relu"),
        Dense(1, activation="sigmoid")
    ])
    return model


student_model = build_student_model(
    input_shape=(X_train_cnn.shape[1], 1)
)

class Distiller(tf.keras.Model):
    def __init__(self, student, teacher, temperature=5.0, alpha=0.5):
        super().__init__()
        self.student = student
        self.teacher = teacher
        self.temperature = temperature
        self.alpha = alpha
        self.student_loss_fn = tf.keras.losses.BinaryCrossentropy()
        self.distillation_loss_fn = tf.keras.losses.KLDivergence()
        self.metric = tf.keras.metrics.BinaryAccuracy()

    def compile(self, optimizer):
        super().compile()
        self.optimizer = optimizer

    def train_step(self, data):
        x, y = data

        teacher_preds = self.teacher(x, training=False)

        with tf.GradientTape() as tape:
            student_preds = self.student(x, training=True)

            student_loss = self.student_loss_fn(y, student_preds)

            distill_loss = self.distillation_loss_fn(
                tf.nn.sigmoid(teacher_preds / self.temperature),
                tf.nn.sigmoid(student_preds / self.temperature)
            )
            total_loss = (
                self.alpha * student_loss +
                (1 - self.alpha) * distill_loss
            )

        grads = tape.gradient(total_loss, self.student.trainable_variables)
        self.optimizer.apply_gradients(
            zip(grads, self.student.trainable_variables)
        )

        self.metric.update_state(y, student_preds)
        return {"loss": total_loss, "accuracy": self.metric.result()}

    def test_step(self, data):
        x, y = data
        preds = self.student(x, training=False)
        loss = self.student_loss_fn(y, preds)
        self.metric.update_state(y, preds)
        return {"loss": loss, "accuracy": self.metric.result()}

distiller = Distiller(
    student=student_model,
    teacher=teacher_model,
    temperature=5.0,   # softer probability distributions AC
    alpha=0.5          # balance hard vs soft targets AC
)

distiller.compile(
    optimizer=Adam(learning_rate=0.001)
)

distiller.fit(
    X_train_cnn, y_train,
    validation_data=(X_val_cnn, y_val),
    epochs=20,
    batch_size=32
)

student_model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)
```

```
student_loss, student_acc = student_model.evaluate(
    X_test_cnn, y_test
)

print("Distilled Student Test Accuracy:", student_acc)

#quantisation of distilled model AC
converter = tf.lite.TFLiteConverter.from_keras_model(student_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

tflite_student = converter.convert()

with open("student_distilled_quantised.tflite", "wb") as f:
    f.write(tflite_student)

print("Final distilled & quantised student model saved.")
# load quantised distilled model AC
interpreter = tf.lite.Interpreter(
    model_path="student_distilled_quantised.tflite"
)
interpreter.allocate_tensors()

# get input/output details AC
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

print("Input details:", input_details)
print("Output details:", output_details)

WINDOW_SIZE = 2500   # one ECG segment (as used in training)
STRIDE = 250         # overlap (90%)



def run_streaming_inference(ecg_signal, window_size=2500, stride=250):
    predictions = []
    inference_times = []

    for start in range(0, len(ecg_signal) - window_size + 1, stride):
        window = ecg_signal[start:start + window_size]
        window = window.reshape(1, window_size, 1).astype(np.float32)

        start_time = time.perf_counter()
        interpreter.set_tensor(input_details[0]['index'], window)
        interpreter.invoke()
        output = interpreter.get_tensor(output_details[0]['index'])[0][0]

        end_time = time.perf_counter()

        predictions.append(output)
        inference_times.append(end_time - start_time)

    return np.array(predictions), np.array(inference_times)



# Take one ECG sample from test set
sample_ecg = X_test[0]   # shape (2500,)

preds, times = run_streaming_inference(sample_ecg)

print("Mean inference time (µs):", times.mean() * 1e6)
print("Max inference time (µs):", times.max() * 1e6)



def apnea_decision(predictions, threshold=0.5, required_ratio=0.6):
    binary_preds = predictions > threshold
    ratio = binary_preds.mean()
    return ratio > required_ratio

apnea_detected = apnea_decision(preds)

print("Apnoea detected:", apnea_detected)



model_size_kb = os.path.getsize(
    "student_distilled_quantised.tflite"
) / 1024

print(f"Final model size: {model_size_kb:.2f} KB")

print(f"Avg latency: {times.mean()*1000:.2f} ms")
print(f"95th percentile latency: {np.percentile(times, 95)*1000:.2f} ms")
```

```
#table of compression and performance summary
results_df = pd.DataFrame({
    "Model": [
        "Pruned CNN",
        "Pruned + Quantised",
        "Distilled + Quantised"
    ],
    "Accuracy": [
        acc,
        tflite_acc,
        student_acc
    ],
    "Model Size (KB)": [
        get_file_size_kb("cnn_sleep_apnea_pruned.keras"),
        get_file_size_kb("cnn_sleep_apnea_pruned_quant.tflite"),
        model_size_kb
    ],
    "Avg Latency (ms)": [
        np.nan,   # not measured for full Keras
        np.nan,
        times.mean() * 1000
    ]
})

print(results_df)
```

15 Project Diary

10 Sept: Refined initial idea focusing on ECG-based sleep apnoea detection.

11 Sept: Conducted background research on sleep apnoea physiology and ECG-based detection methods.

12 Sept: Researched neural networks and CNNs for biomedical signal processing.

13 Sept: Explored relevant datasets and identified the ECG Sleep Apnoea dataset.

14 Sept: Downloaded and inspected dataset structure and class labels.

15 Sept: Cleaned dataset and verified feature dimensions and label balance.

16 Sept: Planned Iteration 1 baseline model architecture.

17 Sept: Implemented data loading and preprocessing pipeline.

18 Sept: Implemented train/validation/test splitting strategy.

19 Sept: Added feature normalisation using standard scaling.

20 Sept: Implemented baseline fully-connected neural network.

21 Sept: Trained baseline model and monitored convergence.

22 Sept: Evaluated baseline accuracy on the test set.

23 Sept: Analysed baseline results and noted limitations.

24 Sept: Began writing Iteration 1 methodology section.

25 Sept: Wrote Iteration 1 testing and evaluation section.

26 Sept: Refined baseline results tables and summaries.

27 Sept: Researched CNN architectures for time-series ECG data.

28 Sept: Planned transition to CNN-based model for Iteration 2.

29 Sept: Sketched CNN architecture with 1D convolutions.

30 Sept: Prepared codebase for CNN input reshaping.

1 Oct: Implemented CNN model with Conv1D and pooling layers.

2 Oct: Debugged CNN input shapes and training issues.

3 Oct: Trained CNN model and compared performance to baseline.

4 Oct: Added validation monitoring and dropout regularisation.

5 Oct: Evaluated CNN model on test data.

6 Oct: Analysed CNN improvements over baseline.

7 Oct: Researched neural network pruning techniques.

8 Oct: Integrated TensorFlow Model Optimization Toolkit.

9 Oct: Implemented structured weight pruning for CNN layers.

10 Oct: Trained pruned CNN with polynomial decay schedule.

11 Oct; Evaluated pruned CNN accuracy and parameter reduction.

12 Oct: Compared pruned CNN performance to unpruned CNN.

13 Oct: Generated training convergence plots for CNN.

14 Oct: Created confusion matrix and prediction confidence plots.

15 Oct: Began writing Iteration 2 methodology section.

16 Oct: Documented pruning approach and design rationale.

17 Oct: Updated results section with CNN metrics.

18 Oct: Researched post-training quantisation for edge deployment.

19 Oct: Implemented TFLite dynamic range quantisation.

20 Oct: Converted pruned CNN to quantised TFLite model.

21 Oct: Evaluated quantised model accuracy.

22 Oct: Measured model size reduction after quantisation.

23 Oct: Compared accuracy vs size trade-off.

24 Oct: Began Iteration 3 planning focused on deployment realism.

25 Oct: Researched knowledge distillation techniques.

26 Oct: Designed teacher-student CNN architecture.

27 Oct: Implemented student CNN model.

28 Oct: Implemented custom distillation training loop.

29 Oct: Trained distilled student model.

30 Oct: Evaluated distilled model accuracy.

31 Oct: Quantised distilled student model to TFLite.

1 Nov: Compared pruned, quantised, and distilled models.

2 Nov: Generated compression and performance summary tables.

3 Nov: Measured inference latency using TFLite interpreter.

4 Nov: Implemented sliding-window streaming inference.

5 Nov: Tested windowed ECG inference on test signals.

6 Nov: Designed apnoea decision logic using ratio thresholds.

7 Nov: Implemented apnoea decision function.

8 Nov: Simulated continuous ECG stream inference.

9 Nov: Added latency tracking and performance logging.

10 Nov: Analysed real-time inference feasibility.

11 Nov: Researched Raspberry Pi deployment requirements.

12 Nov: Prepared SD card and OS for Raspberry Pi.

13 Nov: Attempted TensorFlow Lite runtime installation.

14 Nov: Debugged Python version and package compatibility issues.

15 Nov: Investigated 32-bit vs 64-bit Raspberry Pi constraints.

16 Nov: Attempted virtual environment workaround.

17 Nov: Documented deployment challenges encountered.

18 Nov: Decided to simulate live deployment in software.

19 Nov: Implemented continuous live monitoring loop.

20 Nov: Added console-based apnoea alerts.

21 Nov: Tested continuous monitoring behaviour.

22 Nov: Analysed false positive behaviour in live simulation.

23 Nov: Adjusted decision thresholds conceptually.

24 Nov: Began writing Iteration 3 methodology section.

25 Nov: Wrote testing and evaluation for compressed models.

26 Nov: Created ROC curve and AUC analysis.

27 Nov : Finalised confusion matrix interpretation.

28 Nov: Drafted results section structure.

29 Nov: Populated results section with plots and tables.

30 Nov: Refined accuracy, size, and latency discussion.

1 Dec :Wrote discussion section interpreting results.

2 Dec: Linked compression effects to model behaviour.

3 Dec: Added comparison between CNN and compressed CNNs.

4 Dec: Drafted limitations and errors section.

5 Dec: Refined limitations related to deployment simulation.

6 Dec: Wrote threats to validity section.

7 Dec: Drafted future work and areas of improvement.

8 Dec: Added references to edge-AI and compression literature.

9 Dec: Wrote conclusions section.

10 Dec: Proofread full report draft.

11 Dec: Refined technical explanations for clarity.

12 Dec: Edited figures and captions.

13 Dec: Created poster outline.

14 Dec: Condensed background section for poster.

15 Dec: Adapted methodology for poster format.

16 Dec: Selected key results for poster inclusion.

17 Dec: Designed compression comparison visuals.

18 Dec: Drafted poster discussion points.

19 Dec: Finalised poster layout.

20 Dec: Reviewed poster for clarity and consistency.

21 Dec: Made final corrections to report.

22 Dec: Checked references and citations.

23 Dec: Final formatting and submission preparation.

24 Dec: Backup of all code and documentation.

26 Dec: Light review of report and poster.

29 Dec: Prepared appendix and code listings.

30 Dec: Final proofread.

3 Jan: Final poster export.

4 Jan: Final report export.